

DICIEMBRE DE 2024

# ENTREGA FINAL

## PROGRAMACIÓN CONCURRENTES Y DISTRIBUIDA



Alejandro González Flórez

Ivan Junior Echeverri

Javier Roberto Lopez

# 1. Introducción

En la clase de **Programación Concurrente y Distribuida - 2024-2**, se ha desarrollado un proyecto final titulado **"Portal de Búsqueda de Genes en el Genoma de Uvas"**, orientado a resolver un problema real en la industria vitivinícola chilena.

Este proyecto responde a la necesidad de una de las empresas líderes en producción de vino en Chile, que ha invertido en investigación bioinformática para identificar genes que mejoren la calidad de las uvas utilizadas en sus vinos.

Los datos generados por los investigadores, correspondientes a los genomas de las variedades de uva **chardonnay** y **cabernet**, son almacenados en archivos de más de 2GB cada uno. Sin embargo, la complejidad de estos datos y su volumen han hecho que las herramientas actuales de consulta sean ineficientes, lo que limita su utilidad para el equipo técnico de la empresa. Por lo tanto, se planteó el desarrollo de un buscador eficiente que permita realizar consultas rápidas y precisas sobre los genes identificados.

[Repositorio](#)

## 2. Objetivo del Proyecto

Diseñar e implementar un sistema de búsqueda eficiente basado en técnicas de **programación concurrente y distribuida**, que permita procesar y consultar grandes volúmenes de datos genómicos de manera rápida y efectiva. El objetivo principal es optimizar el tiempo de respuesta en las consultas y garantizar la escalabilidad del sistema para manejar archivos de gran tamaño.

## 3. Propuestas de Solución

Para poder solucionar el problema con la búsqueda eficiente en grandes volúmenes de datos (con archivos de más de 2GB), se ha buscado implementar un sistema que pueda aprovechar las técnicas de programación concurrente y distribuida, intentando optimizar el tiempo de respuesta y poder escalar el sistema a mediano o largo plazo.

Se ha empezado a revisar el problema con la subida de los archivos grandes, para eso se ha elegido utilizar MongoDB.

MongoDB tiene funcionalidades perfectas para el manejo, almacenamiento y procesamiento de datos, incluyendo la capacidad de manejar grandes volúmenes de datos, así como la funcionalidad que permite decidir los archivos en partes más pequeñas (chunks) y almacenarlos de forma distribuida en la base de datos.

## **a. Subida de Archivo:**

### **i. Subida y Almacenamiento con GridFS:**

GridFS es el sistema de almacenamiento nativo de MongoDB para manejar archivos grandes. Divide los archivos en bloques (por defecto, 255KB cada uno) y los almacena en dos colecciones: fs.files (metadatos) y fs.chunks (datos del archivo).

Las principales ventajas al usar esta estrategia es que puede soportar archivos de cualquier tamaño, se integra nativamente con MongoDB y está muy bien optimizada. Su principal desventaja es la complejidad de su desarrollo y que será más lento en comparación con enfoques diseñados para acceso masivo a datos.

### **ii. Subida con Fragmentación Manual del Archivo:**

En lugar de depender de GridFS, se puede implementar un sistema personalizado para dividir y almacenar el archivo en fragmentos directamente en colecciones de MongoDB usando otros lenguajes de programación como Python, C o Java.

Las principales ventajas son un mayor control sobre la lógica de almacenamiento y la posibilidad de optimizar la lógica de desarrollo en cualquier caso.

Su principal desventaja es que la implementación es más manual y se requiere de una cantidad de pruebas importantes para garantizar su correcto funcionamiento.

### **iii. Subida con Compresión y Almacenamiento en MongoDB:**

Antes de almacenar los datos en MongoDB, el archivo puede ser comprimido para reducir el tamaño y optimizar el almacenamiento. Su principal ventaja es la reducción significativa del espacio de almacenamiento de almacenamiento y la simplificación de al tratar un solo archivo y procesarlo directamente.

Su principal desventaja es el tiempo de compresión/descompresión que afectará considerablemente su velocidad de procesamiento.

#### **iv. Subida con Almacenamiento Híbrido (MongoDB + Sistema de Archivos):**

Almacenar los archivos grandes en un sistema de archivos tradicional y utilizar MongoDB para gestionar los metadatos y las referencias. Su principal ventaja es la mejora de la eficiencia al no sobrecargar MongoDB y la facilidad de migrar entre diferentes servidores.

Su principal desventaja es la dependencia del archivo directamente en el servidor y requerir una sincronización constante entre MongoDB y el sistema de archivos.

#### **v. Subida y Almacenamiento con Divisiones por Documento:**

El archivo se divide en partes más pequeñas y se almacena en MongoDB como una colección de documentos JSON estructurados. Su principal ventaja es la facilidad de consultas específicas dentro del archivo y la buena integración con el modelo de MongoDB.

Su principal desventaja es el incremento en el número de documentos en la base de datos y una mayor complejidad en la construcción del archivo.

### **b. Arquitectura basada en Servicios:**

A continuación se mostrarán todas las opciones barajadas a la hora de elegir nuestra arquitectura basada en principios:

#### **i. Flask (Backend) + Vue.js (Frontend):**

Frontend (Vue.js):

Proporciona una interfaz sencilla y progresiva para los usuarios.  
Consume las APIs REST de Flask para interactuar con los datos de MongoDB.

Backend (Flask):

Actúa como una capa intermedia entre MongoDB y Vue.js.  
Implementa funcionalidades básicas como subida de archivos, validación de datos y consultas a MongoDB.

Ventajas:

Simplicidad para proyectos pequeños o medianos.  
Fácil integración entre Flask y MongoDB.

## ii. **Node.js con Express (Backend) + React (Frontend):**

Frontend (React):

Usa componentes modulares para construir la interfaz.  
Consume APIs de Express para realizar operaciones en MongoDB.

Backend (Express):

Maneja solicitudes HTTP y ejecuta lógica de negocio.  
Utiliza Mongoose para interactuar con MongoDB.

Ventajas:

Ecosistema completamente basado en JavaScript.  
Amplio soporte de bibliotecas y herramientas para Node.js.

## iii. **Spring Boot (Backend) + Thymeleaf o React (Frontend):**

Frontend:

Thymeleaf para aplicaciones monolíticas o React para un frontend independiente.

Backend (Spring Boot):

Ofrece APIs REST o renderiza vistas dinámicas directamente.  
Utiliza Spring Data MongoDB para conectarse a la base de datos.

Ventajas:

Ideal para aplicaciones empresariales con grandes volúmenes de datos.  
Spring Boot es robusto y escalable.

## iv. **FastAPI (Backend) + Angular (Frontend):**

Frontend (Angular):

Construye una aplicación SPA (Single Page Application).  
Usa servicios HTTP para consumir APIs del backend.  
Implementa módulos y componentes para manejar la interacción del usuario, como un formulario para subir archivos y tablas/gráficos para visualizar datos.

Backend (FastAPI):

Expone endpoints REST/GraphQL para las operaciones necesarias:

Subir archivos grandes.

Consultar y filtrar genes almacenados en MongoDB.

Maneja la lógica de negocio, como la validación de datos y la paginación.

Usa librerías como Motor o PyMongo para comunicarse con MongoDB.

Ventajas:

Excelente para aplicaciones modernas, rápidas y escalables.

FastAPI es ligero y soporta asincronía, ideal para manejar solicitudes concurrentes.

Angular permite crear una interfaz dinámica y responsiva.

## **c. Consulta de Datos:**

### **i. Consultas Filtradas y Paginadas con FastAPI**

Estas consultas aprovechan la naturaleza asíncrona de Motor y FastAPI para manejar múltiples solicitudes concurrentes de manera eficiente. Son útiles cuando los datos en MongoDB necesitan ser accedidos con frecuencia y presentan filtros dinámicos y paginación.

### **ii. Consultas Simples con PyMongo:**

Son las consultas más básicas que utilizan PyMongo para interactuar con MongoDB. Ideales para proyectos pequeños o con una complejidad mínima en las consultas.

### **iii. Consultas con Índices:**

Los índices en MongoDB permiten que las consultas sean más rápidas al reducir la cantidad de documentos que el motor necesita escanear. Son cruciales para manejar bases de datos grandes.

## 4. Arquitectura Elegida:

Con el objetivo de desarrollar una solución eficiente y escalable para manejar grandes volúmenes de datos genómicos, facilitando su acceso y consulta por parte de los usuarios. Para lograr esto, es esencial definir una arquitectura que optimice el manejo de datos y garantice una experiencia de usuario fluida.

Teniendo en cuenta nuestra experiencia en diferentes lenguajes de programación, la capacidad de tiempo individual y con el propósito de tener un proyecto que garantice el tener en cuenta los requerimientos específicos del proyecto se ha elegido una Arquitectura basada en el manejo de BD de **MongoDB**, usando como backend una API creada en **FastAPI** y para interfaz de usuario **Angular**.

### a. Eficiencia y Rendimiento:

FastAPI es uno de los frameworks más rápidos para construir APIs gracias a su naturaleza asíncrona.

MongoDB, combinado con índices y consultas optimizadas, asegura tiempos de respuesta rápidos incluso con grandes volúmenes de datos genómicos.

### b. Flexibilidad y Escalabilidad:

MongoDB soporta datos no estructurados, lo que es ideal para almacenar información genómica con estructuras complejas o variables.

Angular permite construir una interfaz moderna y adaptable, capaz de crecer con las necesidades del proyecto.

### c. Experiencia de Usuario:

Angular ofrece una experiencia interactiva y fluida mediante su arquitectura basada en componentes.

La integración con paginación, filtros avanzados y visualización en tiempo real mejora la experiencia del usuario final.

### d. Facilidad de Desarrollo y Mantenimiento:

FastAPI tiene una curva de aprendizaje moderada, con documentación clara y soporte para tipado estático con Python.

Angular es ampliamente utilizado en la industria, lo que facilita encontrar talento o soporte.

### **e. Asincronía:**

La capacidad asíncrona de FastAPI y Motor permite manejar múltiples solicitudes simultáneamente, crucial para sistemas de búsqueda en tiempo real.

### **f. Soporte para Grandes Volúmenes de Datos:**

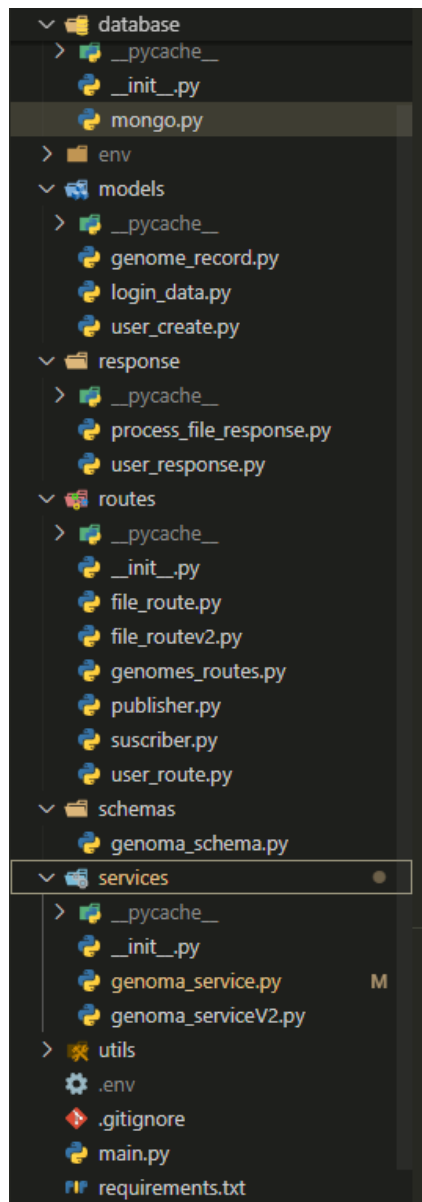
MongoDB es altamente escalable y puede manejar archivos grandes o grandes cantidades de documentos genómicos sin pérdida de rendimiento.

## **5. Explicación Arquitectura:**

### **a. Arquitectura Backend:**

A continuación se explicará la arquitectura del proyecto en la parte backend, esta arquitectura se ha pensado en una organización modular y con separación de responsabilidades, lo cual facilita bastante el desarrollo, mantenimiento y escalabilidad de la aplicación.





### i. main.py

Este archivo suele ser el punto de entrada de la aplicación FastAPI. Aquí se inicializa la instancia de FastAPI, se registran las rutas, se configura la aplicación y se inician servicios o middlewares necesarios.

### ii. requirements.txt:

Archivo donde se listan las dependencias del proyecto

### iii. **.env:**

Archivo de configuración donde se almacenan variables de entorno, como credenciales de la base de datos o claves secretas.

### iv. **Database:**

Este directorio centraliza la configuración de la conexión con la base de datos, **mongo.py** contiene la lógica para conectarse a MongoDB usando un cliente asíncrono (Motor)

### v. **Models:**

Contiene los modelos de datos que se utilizan en la aplicación. Estos modelos describen cómo se estructuran los datos que se envían y reciben.

### vi. **Response:**

Aquí se almacenan las estructuras que definen las respuestas de la API.

### vii. **Routes:**

El directorio más importante de la estructura, ya que aquí se definen los endpoints de la aplicación.

### viii. **Schemas:**

Define esquemas de validación o estructuras de datos específicos relacionados con los genomas. Este archivo utiliza Pydantic y permite garantizar que los datos enviados y recibidos cumplen con la estructura esperada.

### ix. **Services:**

Aquí se organiza la lógica de negocio que no debe estar directamente en las rutas. **genoma\_service.py** y **genoma\_serviceV2.py** Contienen la lógica específica para interactuar con la base de datos o realizar transformaciones de datos relacionadas con los genomas.

## **x. Utils:**

Este directorio puede contener utilidades y funciones auxiliares que son reutilizables en toda la aplicación (e.g., manejo de fechas, validaciones adicionales, funciones de cifrado, etc.).

## **xi. Beneficios de Esta Arquitectura:**

### **1. Modularidad:**

Cada funcionalidad está separada en módulos específicos, lo que facilita el desarrollo, pruebas y mantenimiento.

### **2. Escalabilidad:**

Es fácil agregar nuevas funcionalidades creando nuevos archivos en routes, services o schemas sin afectar el resto de la aplicación.

### **3. Reutilización de Código:**

La lógica de negocio en services y los modelos en schemas y models pueden ser reutilizados en diferentes rutas.

### **4. Separación de Responsabilidades:**

Routes: Manejan la comunicación HTTP.

Services: Procesan la lógica de negocio.

Database: Gestiona la persistencia de datos.

### **5. Mantenimiento Sencillo:**

La estructura clara facilita encontrar archivos y funciones, reduciendo la complejidad.

### **6. Escalabilidad:**

Puedes dividir módulos en microservicios si el proyecto crece.

## b. Arquitectura Frontend:

La arquitectura utilizada es una Arquitectura Basada en Componentes típica de Angular, que sigue los principios del Framework MVC (Model-View-Controller) en conjunto con servicios y modularización.

Angular se basa en una arquitectura orientada a componentes, donde la interfaz de usuario se descompone en piezas reutilizables y funcionales.

Modelo MVC (en la práctica con Angular):

**Model:** Gestión de los datos y comunicación con servicios externos.

**View:** Representación visual de los datos mediante plantillas HTML.

**Controller:** La lógica implementada dentro de los componentes TypeScript.

### a. Estructura General del Proyecto

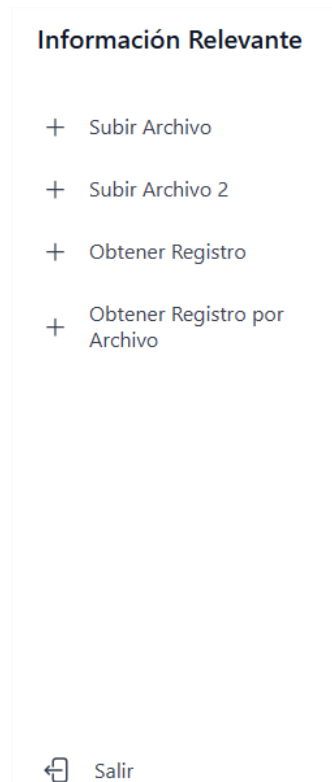
La estructura de carpetas del proyecto fue organizada de manera modular, separando componentes, servicios y estilos de forma independiente.

### b. Componentes Principales

#### **side-menu.component:**

Actúa como un componente principal para la navegación del sistema.

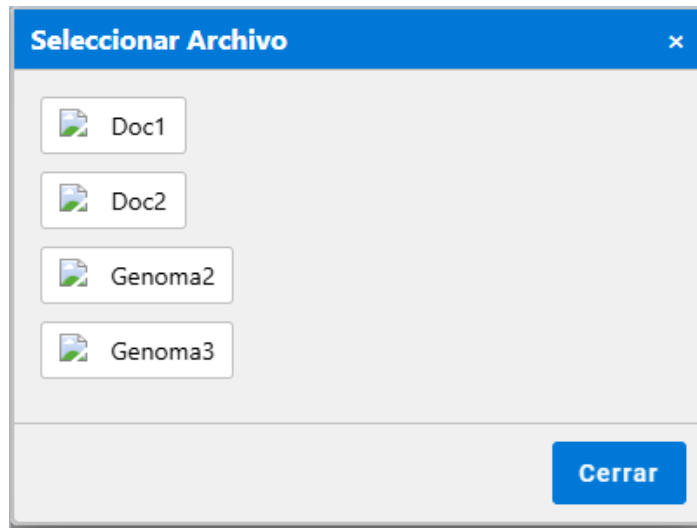
Incluye botones funcionales (Cargar Archivo, Seleccionar Archivo, etc.).



**file-selector.component:**

Componente reutilizable y modular encargado de mostrar los archivos recuperados y permitir su selección.

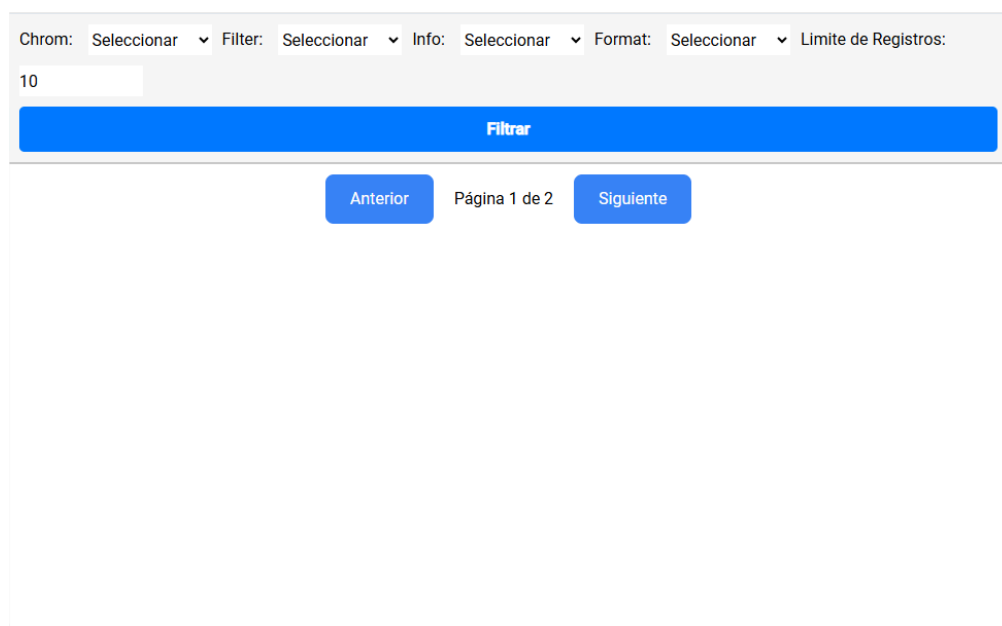
Incluye una interfaz visual organizada similar a una ventana del sistema operativo.



**vista-tabla.component**

Componente reutilizable y modular encargado de mostrar los datos recuperados además de contar con un filtro y paginación.

Incluye una interfaz visual organizada donde se muestran los campos del filtro y un espacio suficiente para mostrar hasta 10 filas..



CHROM	POS	ID	REF	ALT	QUAL	FILTER	INFO	FORMAT
chr1	123456	rs1234	A	T	99	PASS	AC=1;AF=0.5	GT
chr2	789012	rs5678	G	C	95	PASS	AC=2;AF=0.25	GT
chr3	345678	rs91011	T	A	100	PASS	AC=3;AF=0.75	GT
chr1	456789	rs1112	C	G	85	LowQual	AC=1;AF=0.3	GT
chr2	567890	rs1314	A	T	95	PASS	AC=2;AF=0.4	GT
chr3	678901	rs1516	G	C	98	PASS	AC=2;AF=0.6	GT
chr4	789012	rs1718	C	G	92	PASS	AC=1;AF=0.8	GT
chr5	890123	rs1920	T	A	88	PASS	AC=3;AF=0.5	GT
chr6	123789	rs2122	A	T	97	PASS	AC=4;AF=0.3	GT
chr7	234567	rs2324	G	C	93	PASS	AC=2;AF=0.6	GT

Anterior
Página 1 de 2
Siguiente

**popup.component**

Componente reutilizable y modular encargado de mostrar mensajes en la página, utilizado mayormente en el Registro de Usuario.

Registro

Nombre

admin3

Correo

java

Contraseña

.....

Registrar

Cancelar

Correo o contraseña incorrectos

Error

Hubo un error al intentar registrarse. Intente nuevamente.

Cerrar

Registro

Nombre

admin3

Correo

java

Contraseña

.....

Registrar

Cancelar

Correo o contraseña incorrectos

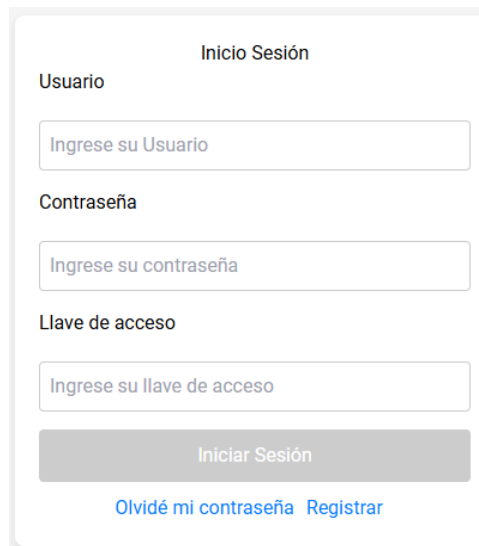
Éxito

Registro exitoso, la llave de acceso fue enviada a su correo electrónico

Cerrar

### **login.component**

Componente utilizado para el formulario de inicio de sesión del usuario, este cuenta con el espacio necesario para el nombre de usuario, contraseña y llave de acceso.



The image shows a login form titled "Inicio Sesión". It contains three input fields: "Usuario" with the placeholder "Ingrese su Usuario", "Contraseña" with the placeholder "Ingrese su contraseña", and "Llave de acceso" with the placeholder "Ingrese su llave de acceso". Below these fields is a grey button labeled "Iniciar Sesión". At the bottom, there are two links: "Olvidé mi contraseña" and "Registrar".

..

### **c. Servicios**

Se implementaron varios servicios estos con el fin de comunicarse con el backend por medio de peticiones HTTP y para la comunicación entre componentes:

- **auth.service**: Servicio encargado del login del usuario, toma las credenciales del usuario (email, contraseña y llave) con estas credenciales hace la petición al

backend y espera su respuesta.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { ConfigServiceService } from './config-service.service';

@Injectable({
  providedIn: 'root'
})
export class AuthService {

  constructor(private http: HttpClient, private configService: ConfigServiceService) {}

  login(credentials: { email: string; password: string; key: string }): Observable<any> {
    return this.http.post(`${this.configService.apiUrl}/login`, credentials);
  }
}
```

- **config.service:** encargado de traer la ruta del backend desde un archivo llamado environment

```
import { Injectable } from '@angular/core';
import { environment } from 'src/assets/environment';

@Injectable({
  providedIn: 'root'
})
export class ConfigServiceService {

  apiUrl = environment.apiUrl;

  constructor() { }
}
```

- **file-upload.service:** este servicio es el que se encarga de hacer la petición al backend cuando se desea subir un archivo, uploadUrl1 está dirigida a la primera estrategia y uploadUrl2 a la segunda.



```

@Injectables({
  providedIn: 'root'
})
export class FileUploadService {

  private uploadUrl = `${environment.apiUrl}/genoma/process_file`; // URL del backend donde se subirá el archivo
  private uploadUrl2 = `${environment.apiUrl}/genomaV2/process_file`;
  constructor(private http: HttpClient) { }

  upload(file: File): Observable<any> {
    const formData = new FormData();
    formData.append('file', file, file.name);

    const headers = new HttpHeaders({
      'enctype': 'multipart/form-data'
    });

    return this.http.post(this.uploadUrl, formData, { headers });
  }

  upload2(file: File): Observable<any> {
    const formData = new FormData();
    formData.append('file', file, file.name);

    const headers = new HttpHeaders({
      'enctype': 'multipart/form-data'
    });

    return this.http.post(this.uploadUrl2, formData, { headers });
  }
}

```

- **popup.service:** este servicio está encargado del formato de los popup de la página (usados en registro)

```

import { MatDialog, MatDialogConfig } from '@angular/material/dialog';
import { PopupComponent } from '../modules/popup/popup.component';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PopupService {
  constructor(private dialog: MatDialog) {}

  showMessage(message: string, success: boolean): Observable<void> {
    const config: MatDialogConfig = {
      width: '300px',
      data: { message, success }
    };
    const dialogRef = this.dialog.open(PopupComponent, config);
    return dialogRef.afterClosed();
  }
}

```

- **app.component:** Es el componente principal de la aplicación donde se muestran todos los html y se muestra en la página.

```
Go to component
<ng-container *ngIf="isAuthenticated(); else loginPage">
  <app-header [selectedFileName]="selectedFileName"></app-header>
  <div class="flex">
    <app-side-menu (filesFetched)="onFilesFetched($event)"></app-side-menu>
    <div class="flex-grow text-black flex-grow p-6 overflow-auto bg-gray-200 w-full mt-0">
      <app-vista-grafo [fileContent]="fileContent"></app-vista-grafo>
    </div>
  </div>
</ng-container>

<ng-template #loginPage>
  <router-outlet></router-outlet>
</ng-template>
```

#### d. Comunicación entre Componentes

La arquitectura adoptó el patrón Input/Output y Observables para garantizar la comunicación eficiente entre componentes:

Observables: Se usaron para manejar la respuesta asíncrona del API.

#### e. Características de la Arquitectura

1. **Modularidad:** Cada funcionalidad está encapsulada en componentes individuales para facilitar la reutilización.
2. **Separación de Responsabilidades:**
  - Los componentes se enfocan en la lógica de presentación.
  - Los servicios manejan las peticiones HTTP y la lógica de negocio.
3. **Escalabilidad:** La arquitectura permite agregar nuevos componentes o servicios sin afectar la funcionalidad existente.
4. **Desacoplamiento:** Los componentes son independientes y pueden comunicarse a través de **@Input** y **@Output** o mediante servicios compartidos.
5. **Responsividad:** Se usaron clases utilitarias de **Tailwind CSS** para asegurar un diseño moderno y adaptable.

## 6. Explicación Estrategia:

La estrategia que se usa es la implementación de un servicio asíncrono para procesar genomas utilizando Motor, una biblioteca asíncrona para interactuar con MongoDB. Aquí te explico los componentes clave y la estrategia general:

### a. MongoDB con Motor:

Motor es una biblioteca asíncrona para trabajar con MongoDB en Python, que aprovecha asyncio para mejorar la eficiencia en operaciones de bases de datos concurrentes. El servicio GenomeProcessorService interactúa con una base de datos MongoDB de forma asíncrona, realizando tareas como obtener datos, procesarlos y posiblemente almacenarlos.

### b. Uso de asyncio:

asyncio es una librería de Python para realizar programación concurrente. En el código, parece que se aprovecharán las operaciones asíncronas para manejar múltiples tareas al mismo tiempo, como la lectura de archivos y la consulta/actualización de datos en la base.

### c. Estrategia de Concurrencia:

La estrategia asíncrona con asyncio y el uso de Motor permite que el programa realice múltiples operaciones sin bloquear el flujo de ejecución. Esto es útil especialmente cuando hay tareas como la lectura de archivos, consultas a bases de datos o procesos largos que pueden realizarse de manera concurrente.

#### Lectura Concurrente de Archivos:

Usar asyncio para manejar múltiples archivos o grandes volúmenes de datos sin bloquear el hilo principal.

#### Manejo de Base de Datos Asíncrona:

Al utilizar Motor (la versión asíncrona de MongoDB), se pueden realizar consultas y actualizaciones en la base de datos sin bloquear la ejecución del programa. Esto es crucial cuando se están realizando varias operaciones de lectura/escritura de manera concurrente.

## **Procesamiento por Lotes:**

El código también menciona un tamaño de lote `BATCH_SIZE = 10000` y `CHUNK_SIZE = 10000`. Estos valores se usan para dividir el procesamiento de los genomas en bloques más pequeños. Esto es útil para manejar grandes volúmenes de datos de manera más eficiente.

La estrategia no es paralela, El código usa `asyncio` y `Motor`, que están diseñados para la concurrencia, no el paralelismo:

`asyncio` permite que el programa gestione múltiples tareas al mismo tiempo sin necesidad de múltiples hilos o procesos. Sin embargo, todas las tareas se ejecutan dentro de un solo hilo y un solo núcleo, lo que significa que solo hay una ejecución en serie a nivel de CPU, pero con la habilidad de alternar rápidamente entre tareas para simular la simultaneidad.

`Motor` es una biblioteca asíncrona para interactuar con MongoDB, lo que significa que las operaciones de la base de datos no bloquean el hilo principal mientras esperan una respuesta. Esto permite que el sistema pueda continuar haciendo otras cosas mientras espera la respuesta de MongoDB.

## **7. Análisis:**

Teniendo en cuenta que las pruebas se han hecho con la siguiente configuración de Hardware

- Intel Core I5 de 8 Núcleos y 12 Hilos
- 8 GB de RAM
- SSD 256GB
- 3'558.695 Registros procesados para pruebas

Se ha analizado cómo la diferencia entre la cantidad de procesos afecta en el rendimiento del procesamiento de los archivos `.vcf`

Número de Procesos	Tiempo Total (min)	Velocidad (Líneas/Segundo)	Uso de CPU
1	11.28	5262	Bajo (~25%)
2	13.48	4400	Medio (~50%)
4	10.34	5738	Alto (~85%)
8	10.12	5862	Máximo (~100%)

#### a. 1 Proceso:

```
INFO:root:Starting processing of C:\Users\ARAGO~1\AppData\Local\Temp\tmpgonkkmjp_cabernetSauvignon-001.vcf
INFO:root:Start time: 2024-12-16 00:09:30.213318
Índices creados con éxito
INFO:root:Total lines to process: 3560000
INFO:root:Number of chunks: 356
INFO:root:Processing completed at: 2024-12-16 00:20:46.738975
INFO:root:Total processing time: 11.28 minutes
INFO:root:Average processing speed: 5262 lines/second
INFO:root:Number of processes used: 1
INFO:root:Chunk size: 10000
INFO: 127.0.0.1:53651 - "POST /genoma/process_file HTTP/1.1" 200 OK
```

Tiempo Total: 11.28 minutos.

Velocidad: 5262 líneas/segundo.

Uso de CPU: Solo un núcleo es utilizado, lo que limita bastante el rendimiento final del aplicativo.

Aunque el tiempo no es el peor, no se aprovecha la capacidad multicore del procesador. El tiempo fue más eficiente que con 2 procesos, aunque con una menor velocidad de líneas procesadas/segundo.

#### b. 2 Procesos:

```
INFO:root:Starting processing of C:\Users\ARAGO~1\AppData\Local\Temp\tmpbq4ikrqo_cabernetSauvignon-001.vcf
INFO:root:Start time: 2024-12-15 23:34:20.699188
Índices creados con éxito
INFO:root:Total lines to process: 3560000
INFO:root:Number of chunks: 356
INFO:root:Processing completed at: 2024-12-15 23:47:49.696774
INFO:root:Total processing time: 13.48 minutes
INFO:root:Average processing speed: 4400 lines/second
INFO:root:Number of processes used: 2
INFO:root:Chunk size: 10000
INFO: 127.0.0.1:52350 - "POST /genoma/process_file HTTP/1.1" 200 OK
```

Tiempo Total: 13.48 minutos (peor resultado).

Velocidad: 4400 líneas/segundo.

Uso de CPU: Alrededor del 50% (uso de 2 núcleos/hilos).

Se observa overhead adicional, probablemente causado por el sistema operativo al administrar 2 procesos, pero sin mejorar el procesamiento.

2 procesos muestran un impacto significativo en el rendimiento: tiempo mayor y velocidad menor.

### c. 4 Procesos:

```
INFO:root:Starting processing of C:\Users\ARAGO~1\AppData\Local\Temp\tmpf5y02en0_cabernetSauvignon-001.vcf
INFO:root:Start time: 2024-12-15 23:20:20.828738
Índices creados con éxito
INFO:root:Total lines to process: 3560000
INFO:root:Number of chunks: 356
INFO:root:Processing completed at: 2024-12-15 23:30:41.212147
INFO:root:Total processing time: 10.34 minutes
INFO:root:Average processing speed: 5738 lines/second
INFO:root:Number of processes used: 4
INFO:root:Chunk size: 10000
INFO: 127.0.0.1:51765 - "POST /genoma/process_file HTTP/1.1" 200 OK
```

Tiempo Total: 10.34 minutos.

Velocidad: 5738 líneas/segundo.

Uso de CPU: Alto (cerca del 85%).

Con 4 procesos, se logra un equilibrio óptimo entre uso de CPU y rendimiento, aprovechando los 4 núcleos físicos del procesador i5.

Con 4 procesos, se mantiene una buena eficiencia, pero con una ligera reducción debido a menos hilos disponibles.

### d. 8 Procesos:

```

INFO: Started server process [8272]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:50969 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:50969 - "GET /openapi.json HTTP/1.1" 200 OK
INFO:root:Starting processing of C:\Users\ARAGO~1\AppData\Local\Temp\tmpa0a4e88y_cabernetSauvignon-001.vcf
INFO:root:Start time: 2024-12-15 22:58:58.170728
INFO:root:Found 21 sample columns
Índices creados con éxito
INFO:root:Total lines to process: 3560000
INFO:root:Number of chunks: 356
INFO:root:Processing completed at: 2024-12-15 23:09:05.491281
INFO:root:Total processing time: 10.12 minutes
INFO:root:Average processing speed: 5862 lines/second
INFO:root:Number of processes used: 8
INFO:root:Chunk size: 10000
INFO: 127.0.0.1:50982 - "POST /genoma/process_file HTTP/1.1" 200 OK

```

Tiempo Total: 10.12 minutos (mejor resultado).

Velocidad: 5862 líneas/segundo.

Uso de CPU: Máximo (100%), gracias al uso de 8 hilos virtuales.

La mejor velocidad se alcanza al utilizar todos los hilos del procesador. Sin embargo, el incremento respecto a 4 procesos es marginal (solo un 2% de mejora), posiblemente debido a la saturación del CPU y la memoria RAM (8 GB).

Con esta cantidad de procesos se muestra una mayor eficiencia debido a una paralelización efectiva y la capacidad de dividir el trabajo en 356 chunks de tamaño 10,000 líneas.

## e. Conclusiones:

La velocidad máxima se logra con 8 procesos: 5862 líneas/segundo y va disminuyendo proporcionalmente con el número de procesos:

8 procesos: 5862 líneas/segundo.

4 procesos: 5738 líneas/segundo.

2 procesos: 4400 líneas/segundo.

1 proceso: 5262 líneas/segundo

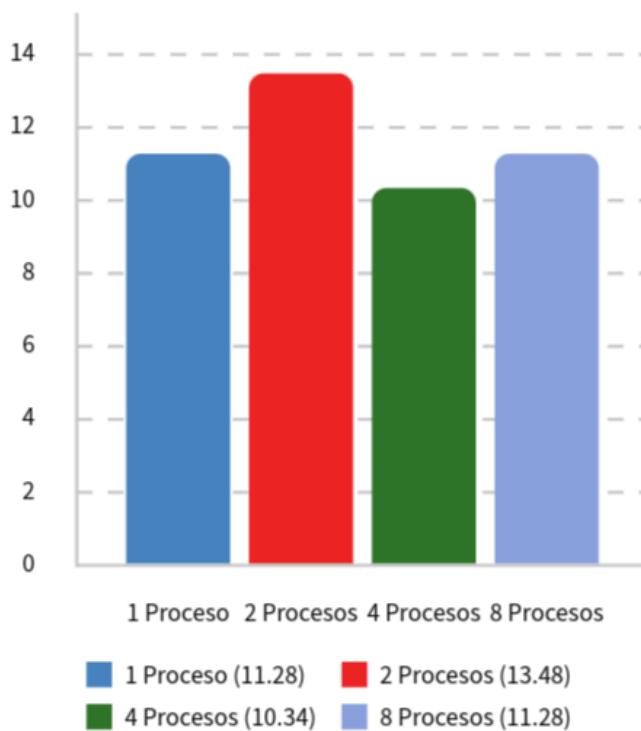
El uso de 4 procesos mantiene una relación eficiencia/recursos equilibrada, obteniendo casi el mismo tiempo de procesamiento que con 8 procesos, esta cantidad ofrece un desempeño cercano al óptimo con menor uso de recursos, lo cual puede ser ideal para sistemas con menos núcleos.

La caída de rendimiento con 2 procesos podría deberse a overhead adicional generado por el manejo de múltiples hilos/procesos sin una cantidad suficiente de núcleos para aprovechar la paralelización, el rendimiento con 2 procesos es notablemente peor, indicando que existe un umbral donde la paralelización pierde eficiencia debido a sobrecarga en la gestión de procesos.

La ejecución con 1 proceso es más eficiente que con 2 procesos, lo cual sugiere que en ciertos casos, menos hilos son preferibles a una mala distribución de la carga.

El tamaño de chunk constante (10,000 líneas) es perfecto para dividir el trabajo entre procesos sin generar overhead innecesario, pero podríamos reducir el tamaño para que el sistema podría experimentar mayor overhead al dividir demasiados fragmentos.

### Gráfica de Tiempo Total vs Número de Procesos



### Gráfica de Velocidad Promedio vs Número de Procesos



