TALLER N° 2

Nombres de los integrantes:	Kevin Asmal, David Delgado, Gabriel López y Marcelo Pareja
Docente:	Mgt. Jenny Alexandra Ruiz Robalino
Fecha:	24/11/2025
NRC:	27835

TALLER 2

1. Objetivo del Taller

Construir una aplicación CRUD de Estudiante (ID, nombres, edad) utilizando 2 arquitecturas, una arquitectura de 3 capas y el patrón de diseño Modelo–Vista–Controlador (MVC), tras ello realizar una pequeña modificación de la arquitectura MVC que incluya el patrón Singleton en la capa de datos y comparar este funcionamiento.

2. Patrón MVC normal

Modelo Vista Controlador (MVC) es un patrón de diseño que separa 3 capas, la capa de datos, donde se definen los objetos abstractos, la capa de lógica de negocios, que son los algoritmos y procesos que modifican, guardan, editan y crean los datos en el sistema y finalmente la capa de presentación, donde se presenta visualmente y con una interacción simple los datos, en este caso el controlador será el que permita manejar todo lo que se realice en el sistema al poder utilizar los servicios que un usuario solicita desde la vista para interactuar con los datos que necesita.

2.1. Código Fuente

Enlace al repositorio: https://github.com/GabMlopez/Trabajos_AyD

2.2. Arquitectura por capas

El proyecto se estructuró siguiendo una arquitectura de 3 capas:

- Capa de datos
- Capa de lógica de negocio
- Capa de presentación

Capa de Datos

Esta capa se encarga de modelar e implementar las clases y objetos que se usarán para almacenar, modificar y compartir información dentro el sistema; en este caso, la información modelada y almacenada es la de los estudiantes.

La capa de datos puede subdividirse en 2 partes. Los modelos, que definen la estructura que tendrá la información, y los repositorios, que almacenan la información en base a las estructuras del modelo, la comparten con el resto del sistema, y manejan las operaciones básicas de tratamiento de los datos: Crear, Leer, Modificar y Eliminar.

Modelos

Aquí se encuentra la clase "*Estudiante*", que define la estructura e información de los estudiantes en el sistema. Cada estudiante cuenta con 3 atributos

- ID
- Nombre
- Edad

Para cada atributo, se tiene su respecivo getter y setter.

Repositorios

Aquí se encuentra la clase "*EstudianteRepositorio*", que representa la lista de estudiantes que maneja el sistema mediante un atributo de lista "lista_estudiantes".

Además, posee 4 métodos que interactúan con la información contenida.

- *agregar()* – Recibe y agrega un nuevo objeto "Estudiante" a la lista.
- *obtener_todos()* – Retorna la lista completa de estudiantes
- *buscar_por_id()* – Recibe un ID numérico, y retorna la información del estudiante al que corresponde.
- *eliminar_por_id()* – Recibe el ID de un estudiante y lo elimina de la lista
- *modificar_por_id()* – Recibe el ID de un estudiante, junto con nuevos datos y modifica la información estudiante con el ID correspondiente.

Capa de Lógica de Negocio

Esta capa es responsable de implementar y asegurar el cumplimiento de las reglas y lógica de negocio que utiliza el sistema. Contiene varias funciones y métodos para validar información, o realizar modificaciones específicas según el propósito y lógica del sistema. Además, permite que la capa de presentación acceda a estos métodos, y, por tanto, que el usuario también pueda acceder a ellos desde la interfaz visual. Los métodos que contiene son:

En esta capa se encuentra la clase “EstudianteServicio”, que contiene funciones para validar el ingreso de datos, y realizar las conversiones al tipo adecuado antes de enviarlos a la capa de datos.

Capa de Presentación

Esta capa contiene la interfaz gráfica que se encarga de interactuar directamente con el usuario. La interfaz está compuesta por 3 botones, cada uno abre un formulario para realizar la inserción, modificación o eliminación de estudiantes, y una tabla que muestra todos los estudiantes del repositorio como se muestra en la figura 1.

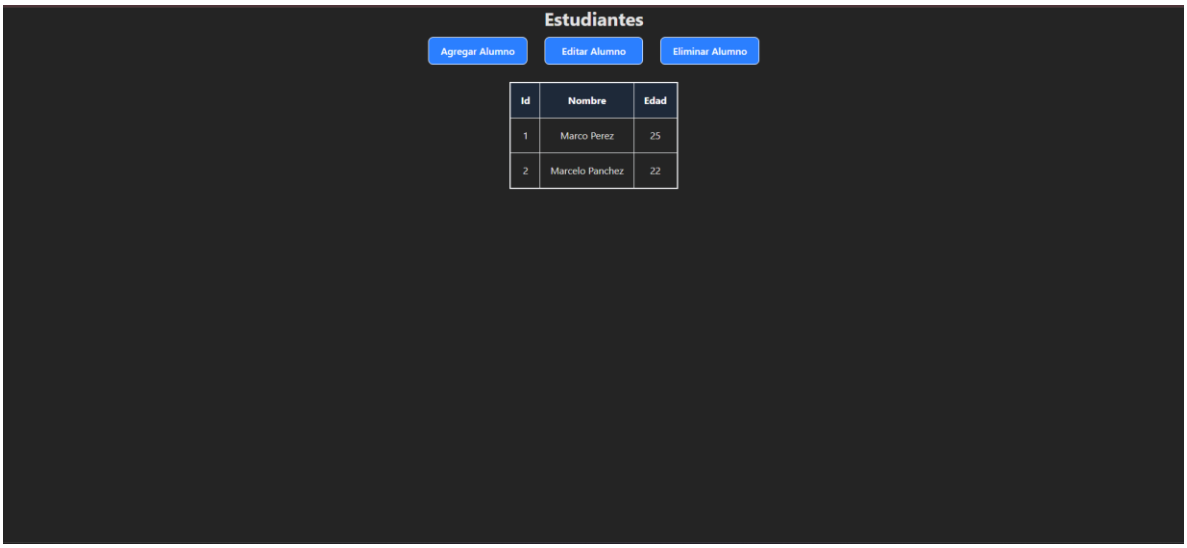

Figura 1: Vista Agregar Estudiante

2.3. Estructura del Proyecto

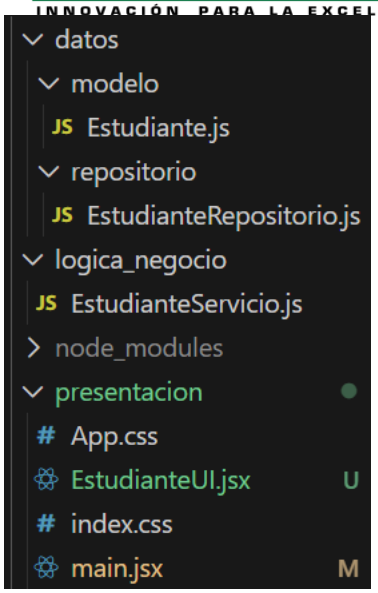

Figura 2: Estructura de Carpetas en formato MVC

2.4. Diagrama de Arquitectura

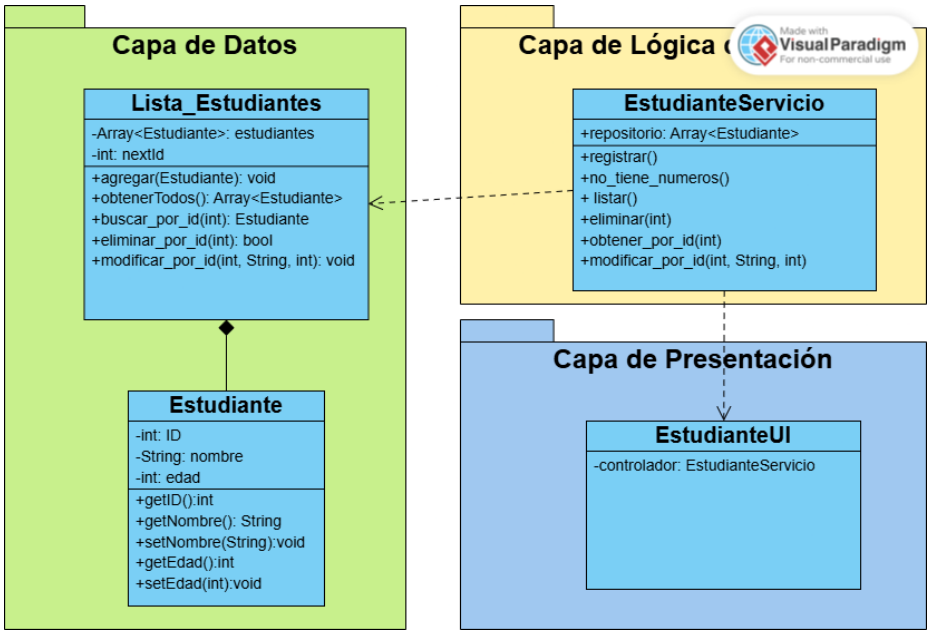

Figura 3: Arquitectura de la interacción entre capas del MVC

2.5. Análisis

2.5.1. Interacción del Sistema

Figura 4: Vista Agregar Estudiante

Figura 5: Vista Editar Estudiante

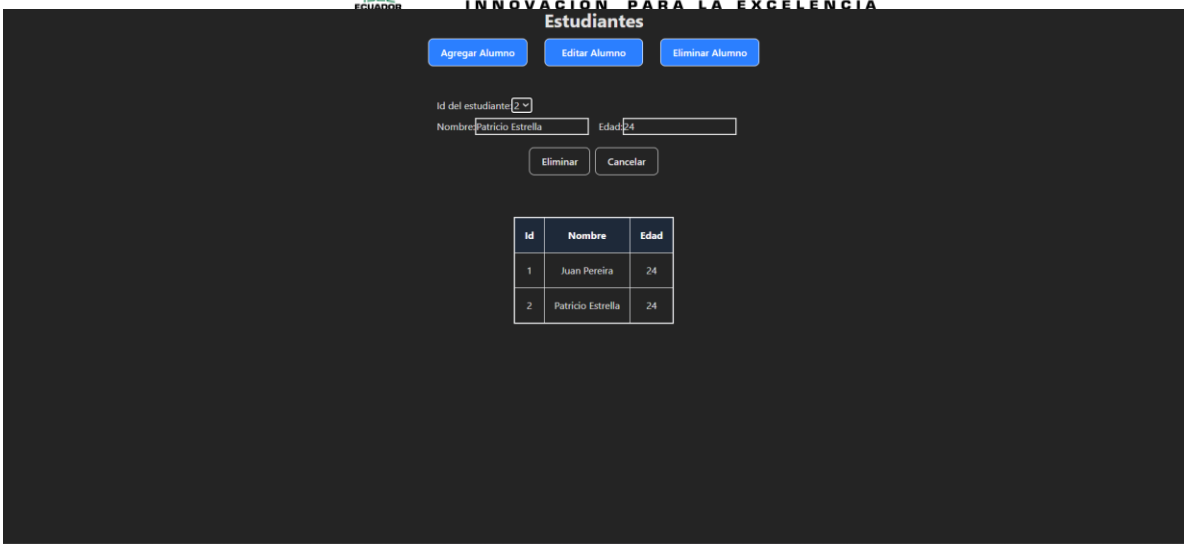

Figura 6: Vista Eliminar Estudiante

2.5.2. Análisis

En las figuras 4, 5 y 6 se muestra el funcionamiento de los servicios de creación, edición y eliminación respectivamente, el listado se realiza automáticamente tras cada acción para observar el cambio en cada acción, esto es posible al poder separar el objeto abstraído (estudiante) de los procesos que manipulan los datos (lógica de negocio), esto no solo permite un mejor control de los componentes en el código, ayuda a la mantenibilidad y resiliencia ante el cambio gracias a esta separación, que en caso de algún cambio en el sistema, se afectara uno o un número pequeño de módulos que será más fácil de controlar, revisar y reparar.

3. Modelo MVC e Implementación Singleton

```
constructor() {
  if (Lista_Estudiantes.instancia) {
    return Lista_Estudiantes.instancia;
  }
  this.estudiantes = [];
  this.siguienteId = 1;
  Lista_Estudiantes.instancia = this;
}
```

Figura 7: Constructor con Singleton

```
static obtenerInstancia() {
  if (!Lista_Estudiantes.instancia) {
    Lista_Estudiantes.instancia = new Lista_Estudiantes();
  }
  return Lista_Estudiantes.instancia;
}
```

Figura 8: Creación de única instancia en Singleton

Item	MVC	MVC+ Singleton
Firura7: this.estudiantes[ ]	Si se crea una nueva instancia de ***Lista_Estudiantes*** cada vez que llega una petición al servidor, el array se reiniciaría a vacío []. Un usuario agregaría un estudiante, pero al intentar listarlos, obtendría una lista vacía.	Garantiza que todas las partes del código (controladores, servicios, tests) accedan exactamente a la misma instancia y, por lo tanto, al mismo array de datos.

1. Estado Global Compartido

Se ha convertido el repositorio en un Estado Global. Esto permite que cualquier archivo que haga import ... from 'lista_estudiantes.js' vea los cambios realizados por otros archivos inmediatamente. Es muy útil para prototipos rápidos o aplicaciones pequeñas donde no hay una base de datos real (como SQL o MongoDB).

3. Eficiencia de Memoria

Aunque en este caso es leve, el Singleton asegura que solo exista un objeto Lista_Estudiantes en la memoria de tu servidor, en lugar de crear y destruir objetos repositorio constantemente con cada solicitud HTTP.

4. Limitación de Escalabilidad (Punto a tener en cuenta)

El impacto negativo o limitante es que esta persistencia solo funciona en un solo proceso. Si en el futuro quisieras usar el modo "Cluster" de Node.js (usar varios núcleos del CPU) o tener varios servidores balanceados, este Singleton fallaría, ya que cada proceso tendría su propia copia del Singleton y los datos no se compartirían entre ellos. Para solucionar eso, necesitarías una base de datos externa real.

4. Comparación de MVC y Singleton

Item	MVC	Singleton
Que problema resuelve	Distribución de las responsabilidades entre capas de un sistema	Generación de una sola instancia para evitar problemas en control de datos
Capa en que se utiliza	Con esta se desarrolla las 3 capas de datos, lógica de negocio y vista	Se utiliza en la capa de datos
Como influye en el mantenimiento	Separa en módulos lo que permite un mejor control a cambios y resiliencia a errores	Evita que se generen varias instancias de un modelo de la capa de datos para un mejor control de los datos
Como evita fallas en el diseño	Al separar por módulos y capas se puede revisar con mayor precisión que errores tiene cada capa	Esto ayuda en el control de los datos generados con una sola instancia y un solo acceso, evita problemas de rendimiento

Explicar el impacto en la persistencia de datos.

El impacto en la persistencia de datos se evidencia al comparar el funcionamiento del MVC tradicional con la versión que incorpora el patrón Singleton. En el MVC normal, si la capa de datos se instancia cada vez que el sistema ejecuta una operación, el repositorio se reinicia y la información almacenada se pierde, lo que impide mantener un estado consistente entre acciones. Esto significa que los datos no persisten durante la ejecución y cada módulo podría trabajar con listas diferentes sin sincronización.

Al implementar el patrón Singleton en la capa de datos, se garantiza que todo el sistema utilice una única instancia del repositorio. Esto permite que la información se mantenga en memoria durante toda la ejecución y que cualquier cambio realizado sea visible inmediatamente desde cualquier parte del sistema. El repositorio se convierte en un estado global compartido, mejorando la persistencia, coherencia y eficiencia del manejo de datos.

Sin embargo, esta persistencia basada en Singleton solo funciona en aplicaciones que corren en un único proceso. En entornos escalados, con múltiples procesos o servidores, cada proceso tendría su propia copia del repositorio, perdiendo sincronización. Para persistencia real y distribuida, sería necesario utilizar una base de datos externa.

5. Anexos

Enlace al repositorio: https://github.com/GabMlopez/Trabajos_AyD