

# Sprint 1 - Trabajo Práctico 4

---

## Índice

1. Introducción a Express (mini tarea 1 y 2)
2. Introducción a la Arquitectura MVC
3. Un servidor con Arq. MVC, una API Rest de Superheroes V1.0.0

## Índice 1: Introducción a Express

**Objetivo:** Introducir y configurar el framework Express para el desarrollo de aplicaciones web y APIs en Node.js, explorar diferentes tipos de ruteo y realizar actividades prácticas para manejar parámetros en solicitudes GET .

### Teoría:

Express es un framework de servidor web para Node.js que proporciona una infraestructura ligera y flexible para el desarrollo de aplicaciones web y APIs. Su diseño modular y su enfoque en middleware permiten una gran facilidad para gestionar el flujo de solicitudes y respuestas, hacer rutas condicionales, y manejar la lógica de la aplicación de manera eficiente.

- **Conceptos Clave:**

- **Middleware:** Las funciones de middleware en Express permiten interceptar y modificar las solicitudes y respuestas en diferentes puntos del ciclo de vida de una solicitud HTTP. Los middleware son útiles para tareas como logging, autenticación, y manejo de errores. Se pueden aplicar a nivel global para todas las rutas o a nivel específico para rutas individuales.
- **Ruteo:** Express permite definir rutas de manera declarativa. Cada ruta se asocia con una función que maneja las solicitudes para esa ruta específica. Los métodos HTTP (GET, POST, PUT, DELETE) se utilizan para definir cómo se manejan diferentes tipos de solicitudes en cada ruta.
- **Gestión de Solicitudes y Respuestas:** Express simplifica la gestión de solicitudes y respuestas HTTP, proporcionando métodos para enviar respuestas de diferentes tipos (texto, JSON, HTML) y para acceder a los datos enviados en la solicitud.

### Pasos y Ejemplo:

#### 1. Instalación de Express:

Instala Express en tu proyecto usando NPM para añadir el paquete a tus dependencias.

#### Comando para instalar Express:

```
npm install express
```

#### 2. Configuración Básica del Servidor:

Configura un servidor básico en Express creando un archivo `app.js` y definiendo una ruta simple.

#### Contenido de `app.js`:

```
import express from 'express';

// Crear una instancia de Express
const app = express();

// Configurar el puerto en el que el servidor escuchará
const PORT = 3000;

// Ruta básica
app.get('/', (req, res) => {
    res.send('¡Hola, mundo!');
});

// Iniciar el servidor
app.listen(PORT, () => {
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

### 3. Ejemplos de Ruteo en Express:

#### Ruteo Básico:

Define rutas que responden a solicitudes `GET` específicas.

```
// Ruta GET para el home
// Solicitud: http://localhost:3000/
app.get('/', (req, res) => {
    res.send('Página de inicio');
});

// Ruta GET para recibir datos simples
// Solicitud: http://localhost:3000/data
app.get('/data', (req, res) => {
    res.send('Datos recibidos');
});
```

#### Ruteo con Parámetros:

Maneja rutas que contienen parámetros dinámicos en la URL.

```
// Ruta GET con parámetro de ruta
// Solicitud: http://localhost:3000/user/123
app.get('/user/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`Perfil del usuario con ID: ${userId}`);
});

// Ruta GET con múltiples parámetros
// Solicitud: http://localhost:3000/product/electronics/456
```

```
app.get('/product/:category/:id', (req, res) => {
  const { category, id } = req.params;
  res.send(`Categoría: ${category}, ID del producto: ${id}`);
});
```

### Ruteo con Consultas:

Maneja rutas que utilizan parámetros de consulta en la URL.

```
// Ruta GET con parámetro de consulta
// Solicitud: http://localhost:3000/search?q=javascript
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`Resultados de búsqueda para: ${query}`);
});

// Ruta GET con múltiples parámetros de consulta
// Solicitud: http://localhost:3000/filter?type=book&minPrice=10&maxPrice=50
app.get('/filter', (req, res) => {
  const { type, minPrice, maxPrice } = req.query;
  res.send(`Filtrar por tipo: ${type},
            rango de precios: ${minPrice} - ${maxPrice}`);
});
```

### Explicación:

- **Ruteo Básico:** Define cómo manejar diferentes tipos de solicitudes `GET` en distintas rutas.  
`app.get()` se usa para manejar solicitudes `GET`.
- **Ruteo con Parámetros:** Permite capturar partes de la URL que son variables y usarlas en el manejo de la solicitud. Esto es útil para crear rutas dinámicas que responden a diferentes valores.
- **Ruteo con Consultas:** Utiliza parámetros de consulta en la URL para realizar operaciones como filtrado y búsqueda. Estos parámetros se pueden acceder a través de `req.query`.

## Actividades de Aprendizaje

### Actividad 1: Manejo de Parámetros de Ruta

1. **Objetivo:** Configurar un servidor Express que reciba una solicitud `GET` con un parámetro `id` en la URL y muestre ese parámetro en la consola.

#### 2. Pasos:

1. Crea un archivo `server.mjs` (o usa `app.mjs` si lo prefieres).
2. Configura una ruta que capture el parámetro `id` en la URL.
3. Muestra el valor del parámetro `id` en la consola cuando se reciba una solicitud.

### Código:

```
import express from 'express';
```

```

const app = express();
const PORT = 3000;

// Ruta GET con parámetro de ruta
// Solicitud: http://localhost:3000/user/123
app.get('/user/:id', (req, res) => {
    const userId = req.params.id;
    console.log(`ID del usuario recibido: ${userId}`);
    res.send(`Perfil del usuario con ID: ${userId}`);
});

app.listen(PORT, () => {
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
});

```

## Actividad 2: Manejo de Parámetros de Consulta

- Objetivo:** Configurar un servidor Express que reciba una solicitud `GET` con un parámetro de consulta `edad` en la URL y muestre ese parámetro en la consola.

### 2. Pasos:

- Crea un archivo `server.mjs` (o usa `app.mjs` si lo prefieres).
- Configura una ruta que capture el parámetro de consulta `edad`.
- Muestra el valor del parámetro `edad` en la consola cuando se reciba una solicitud.

### Código:

```

import express from 'express';

const app = express();
const PORT = 3000;

// Ruta GET con parámetro de consulta
// Solicitud: http://localhost:3000/profile?edad=30
app.get('/profile', (req, res) => {
    const edad = req.query.edad;
    console.log(`Edad recibida: ${edad}`);
    res.send(`Edad del perfil: ${edad}`);
});

app.listen(PORT, () => {
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
});

```

## Índice 2: Introducción a MVC

En este trabajo, vamos a implementar la arquitectura **Modelo-Vista-Controlador (MVC)** en una aplicación Node.js que gestiona tareas, con la diferencia de que vamos a incluir una **capa de servicio** y una **capa de persistencia** que lee los datos desde un archivo de texto usando el módulo `fs` de Node.js. Además, se utilizará un **nivel de abstracción** para que la capa de persistencia pueda cambiarse fácilmente en el futuro sin afectar la lógica de negocio.

La estructura modular de MVC no solo facilita la separación de responsabilidades, sino que también permite añadir nuevas capas de abstracción como servicios y persistencia para hacer que la aplicación sea más escalable y mantenible a largo plazo. Cada componente tiene un rol bien definido:

- **Modelo:** Representa los datos y la lógica de negocio.
- **Vista:** Presenta los datos al usuario, formateados adecuadamente.
- **Controlador:** Recibe las solicitudes del cliente y coordina la interacción entre el modelo, la capa de servicio, y la vista.
- **Capa de Servicio:** Intermediaria entre el controlador y la capa de persistencia. Contiene la lógica de negocio.
- **Capa de Persistencia:** Se encarga de manejar el acceso a los datos, en este caso, leyendo y escribiendo en un archivo de texto.

### Componentes del MVC: Explicación Teórica con Ejemplo

#### Mini Ejemplo Completo con MVC y Abstracción de Persistencia

Supongamos que tenemos un archivo `tareas.txt` con el siguiente contenido:

```
[  
  { "id": 1, "titulo": "Tarea 1", "descripcion": "Descripción de tarea 1",  
    "completado": false },  
  { "id": 2, "titulo": "Tarea 2", "descripcion": "Descripción de tarea 2",  
    "completado": true },  
  { "id": 3, "titulo": "Tarea 3", "descripcion": "Descripción de tarea 3",  
    "completado": true }  
]
```

### Estructura del Proyecto

```
/ejemplo-mvc  
├── └── /models # Modelos de datos  
│   └── └── tarea.mjs # Modelo de tarea  
├── └── /controllers # Controladores  
│   └── └── tareaController.mjs # Controlador de tarea  
└── └── /services # Servicios de lógica de negocio  
    └── └── tareaService.mjs # Servicio de tarea
```

```

└── /repository           # Repositorio de datos
    └── tareaRepository.mjs # Repositorio de tareas
    └── tareasDataSource.mjs # Fuente de datos de tareas
└── /views                # Vistas (frontend o representaciones)
    └── tareaVista.mjs     # Vista de la tarea
└── server.mjs            # Archivo principal del servidor
└── tareas.txt             # Archivo de texto con tareas

```

## 1. Modelo (Model)

La **capa de modelo** en una aplicación MVC es la encargada de representar la estructura de los datos y las operaciones que se pueden realizar sobre ellos. En este caso, el modelo **Tarea** representa una tarea con atributos como `id`, `titulo`, `descripcion` y `completado`. Además, este modelo contiene métodos que permiten realizar operaciones como marcar una tarea como completada o validar sus atributos.

---

### Archivo: `tarea.mjs` (Modelo de Tarea)

Este archivo define el **modelo Tarea**, encapsulando tanto los atributos como los métodos necesarios para manipular los datos de una tarea.

```

export default class Tarea {
  constructor(id, titulo, descripcion, completado = false) {
    this.id = id; // Identificador único de la tarea
    this.titulo = titulo; // Título de la tarea
    this.descripcion = descripcion; // Descripción de la tarea
    this.completado = completado; // Estado de completado, por defecto es false
  }

  // Método para marcar una tarea como completada
  completar() {
    this.completado = true; // Cambia el estado de completado a true
  }

  // Método para validar que el título de la tarea no esté vacío
  validar() {
    if (!this.titulo || this.titulo.trim() === '') {
      throw new Error('El título de la tarea es obligatorio.');
    }
  }
}

```

---

## Explicación del Código

### 1. Clase Tarea :

- La clase `Tarea` es el modelo que encapsula los atributos de una tarea y sus métodos para interactuar con ellos.
- **Atributos:**
  - `id`: Representa un identificador único para cada tarea.
  - `titulo`: El nombre o título que describe la tarea.
  - `descripcion`: Detalle adicional que describe la tarea.
  - `completado`: Un estado booleano que indica si la tarea ha sido completada o no. Por defecto, se inicializa en `false`.

## 2. Método `completar()`:

- Este método cambia el estado de la tarea a `completada` (`true`). Al llamar a este método, podemos marcar una tarea como terminada.

## 3. Método `validar()`:

- Este método se asegura de que la tarea tenga un título válido. Si el título está vacío o consiste solo en espacios, lanza un error con un mensaje que indica que el título es obligatorio. Esta validación ayuda a evitar que se creen tareas sin información necesaria.
- 

## Justificación del Diseño

### 1. Responsabilidad Única:

- La clase `Tarea` sigue el **Principio de Responsabilidad Única (SRP)** al enfocarse únicamente en la representación y manipulación de los datos relacionados con una tarea. Encapsula tanto los atributos como los métodos que pueden operar sobre ellos.

### 2. Simplicidad:

- Este diseño es simple y directo. Al no usar una interfaz intermedia, el modelo `Tarea` es utilizado directamente por la capa de servicios, lo que simplifica la arquitectura y reduce la sobrecarga de código.

### 3. Modularidad:

- Aunque es un diseño simple, el modelo sigue siendo modular. Las funciones relacionadas con la validación y la manipulación de los datos de la tarea están encapsuladas dentro de la clase `Tarea`, lo que facilita su reutilización y mantenimiento.

### 4. Validación Interna:

- El método `validar()` asegura que los datos sean consistentes antes de que la tarea sea procesada o guardada. Esto agrega una capa de robustez al sistema, evitando que tareas incompletas o incorrectas se almacenen o procesen.

---

## Conclusión

La **capa de modelo (Model)** es crucial para representar y manejar los datos dentro de la aplicación. En este caso, el modelo `Tarea` encapsula los atributos y la lógica necesaria para manipular las tareas, como marcar una tarea como completada o validar sus atributos. Al mantener el modelo sencillo y directo, podemos lograr un diseño modular, fácil de mantener y lo suficientemente flexible para futuras expansiones.

Este enfoque de **interactuar directamente con el modelo** simplifica el sistema al no usar interfaces innecesarias para este nivel de abstracción, lo que facilita el desarrollo y la mantenibilidad del código sin sacrificar flexibilidad o robustez.

---

## 2. Vista (View)

La **capa de vista (View)** es responsable de la presentación de los datos, formateándolos para su entrega al cliente. En este caso, trabajamos con una API que devuelve los datos en formato **JSON**, lo que significa que la vista toma los datos procesados por la capa de control y los formatea en JSON antes de enviarlos de vuelta al cliente.

Esta capa es esencial para:

1. **Separar la lógica de presentación** de la lógica de negocio.
  2. **Garantizar que los datos** se presenten de manera consistente y clara.
  3. **Facilitar la reutilización** de los métodos de presentación en toda la aplicación.
- 

### Archivo: `tareaVista.js` (Capa de Vista para las Tareas)

Este archivo define las funciones necesarias para renderizar las tareas y los mensajes en **JSON**.

```
// Función para renderizar una lista de tareas en formato JSON
export function renderizarListaTareas(tareas) {
    // Formatea el array de tareas en formato JSON con indentación
    return JSON.stringify(tareas, null, 2);
}

// Función para renderizar un mensaje genérico en formato JSON
export function renderizarMensaje(mensaje) {
    // Envuelve un mensaje en formato JSON
    return JSON.stringify({ mensaje }, null, 2);
}

// Función para renderizar una tarea específica en formato JSON
export function renderizarTarea(tarea) {
    // Formatea una tarea individual en formato JSON con indentación
    return JSON.stringify(tarea, null, 2);
```

}

---

## Explicación del Código

### 1. Función `renderizarListaTareas()`:

- Toma un array de tareas y lo convierte a formato JSON usando `JSON.stringify()`, con una indentación de 2 espacios para mejorar la legibilidad. Esta función se utiliza cuando la aplicación necesita devolver una lista de tareas al cliente.

### 2. Función `renderizarMensaje()`:

- Envuelve un mensaje en un objeto JSON y lo formatea con una indentación de 2 espacios. Es ideal para enviar mensajes de éxito, error o cualquier otro tipo de notificación al cliente.

### 3. Función `renderizarTarea()`:

- Convierte una tarea individual en JSON con `JSON.stringify()` y lo formatea. Esta función se utiliza cuando se necesita devolver una tarea específica, como al completar una tarea o al solicitar una tarea por su ID.
- 

## Justificación del Diseño

### 1. Modularidad:

- Cada función es responsable de una tarea específica: formatear listas, tareas individuales o mensajes. Esto sigue el **principio de responsabilidad única**, lo que hace que el código sea fácil de mantener y reutilizar.

### 2. Simplicidad:

- Las funciones son sencillas y directas, limitándose a formatear los datos en JSON. Esto garantiza que la capa de vista no se involucra en la lógica de negocio y se mantiene enfocada en la presentación.

### 3. Consistencia:

- Al centralizar la lógica de formateo, garantizamos que todas las respuestas sean consistentes en cuanto a su formato, lo cual es crucial en una API.
- 

## Conclusión

La **capa de vista (View)** es esencial para manejar cómo se presentan los datos a los clientes. Al estructurar

la vista de manera que se encargue exclusivamente del formateo en JSON, logramos una arquitectura limpia, modular y fácil de mantener, siguiendo los principios de **separación de responsabilidades y modularidad**. Esto permite que el resto de la aplicación esté desacoplada de los detalles de presentación, lo que facilita la escalabilidad y mantenibilidad del sistema.

---

### 3. Controlador (Controller)

El **Controlador** es el intermediario entre el modelo, la vista y las solicitudes del usuario. Su función es recibir las solicitudes HTTP, interactuar con la capa de servicio para procesar los datos, y enviar la respuesta utilizando la vista. La lógica de negocio no debe estar en el controlador, sino en la capa de servicio.

#### Ejemplo Teórico

```
import { listarTareas, listarTareasCompletadas, crearTarea, completarTarea, eliminarTarea } from '../services/tareaService.mjs';
import { renderizarListaTareas, renderizarMensaje } from '../views/tareaVista.mjs';

// Controlador para listar todas las tareas
export function listarTareasController(req, res) {
    const tareas = listarTareas();
    res.send(renderizarListaTareas(tareas));
}

// Controlador para listar solo las tareas completadas
export function listarTareasCompletadasController(req, res) {
    const tareasCompletadas = listarTareasCompletadas();
    res.send(renderizarListaTareas(tareasCompletadas));
}

// Controlador para crear una nueva tarea
export function crearTareaController(req, res) {
    const { id, titulo, descripcion, completado } = req.body;
    crearTarea(id, titulo, descripcion, completado);
    res.send(renderizarMensaje('Tarea creada con éxito'));
}

// Controlador para marcar una tarea como completada
export function completarTareaController(req, res) {
    const { id } = req.params;
    completarTarea(parseInt(id));
    res.send(renderizarMensaje('Tarea marcada como completada'));
}

// Controlador para eliminar una tarea
export function eliminarTareaController(req, res) {
    const { id } = req.params;
    eliminarTarea(parseInt(id));
    res.send(renderizarMensaje('Tarea eliminada con éxito'));
}
```

## Explicación teórica de las funciones en `tareaController.js`

### 1. `listarTareasController(req, res)`

**Función:** Controlador para listar todas las tareas.

#### Explicación teórica:

En este caso, la función actúa como un controlador que maneja la solicitud HTTP `GET` para recuperar todas las tareas. El controlador se encarga de recibir la solicitud, llamar al servicio que obtiene las tareas desde la capa de persistencia, y luego formatear la respuesta en JSON utilizando la vista.

#### Flujo:

- Solicitud `GET` enviada a `/tareas`.
- El controlador llama al servicio para obtener la lista de tareas.
- La respuesta se envía en formato JSON.

### 2. `listarTareasCompletadasController(req, res)`

**Función:** Controlador para listar solo las tareas completadas.

#### Explicación teórica:

Este controlador se encarga de gestionar la solicitud HTTP `GET` para listar únicamente las tareas que están marcadas como completadas. El controlador delega la tarea de filtrar las tareas al servicio, que obtiene los datos de la capa de persistencia, y luego la vista formatea esos datos para enviarlos como respuesta.

#### Flujo:

- Solicitud `GET` enviada a `/tareas/completadas`.
- El controlador solicita al servicio que filtre las tareas completadas.
- La respuesta se envía en formato JSON.

### 3. `crearTareaController(req, res)`

**Función:** Controlador para crear una nueva tarea.

#### Explicación teórica:

Esta función recibe una solicitud `POST` con los datos de una nueva tarea en el cuerpo de la solicitud (`req.body`). El controlador pasa esos datos a la capa de servicio, donde se valida y persiste la nueva tarea. Posteriormente, la vista envía un mensaje de confirmación al cliente.

#### Flujo:

- Solicitud `POST` enviada a `/tareas` con los datos de la tarea.
- El controlador pasa los datos al servicio, que valida y almacena la tarea.
- La vista envía un mensaje de éxito en formato JSON.

#### 4. completarTareaController(req, res)

**Función:** Controlador para marcar una tarea como completada.

##### Explicación teórica:

Este controlador maneja la solicitud `PUT` para actualizar una tarea específica y marcarla como completada. Utiliza el parámetro `id` de la URL para identificar la tarea que debe actualizarse. El servicio se encarga de realizar la actualización, y luego la vista envía un mensaje de confirmación.

##### Flujo:

- Solicitud `PUT` enviada a `/tareas/:id/completar` (por ejemplo, `/tareas/1/completar`).
- El controlador pasa el `id` de la tarea al servicio, que la marca como completada.
- La vista envía un mensaje de éxito.

#### 5. eliminarTareaController(req, res)

**Función:** Controlador para eliminar una tarea.

##### Explicación teórica:

Esta función controla la solicitud `DELETE` para eliminar una tarea específica, utilizando el parámetro `id` de la URL. El controlador delega la lógica de eliminación a la capa de servicio, que luego interactúa con la capa de persistencia para eliminar la tarea. Finalmente, la vista envía una confirmación al cliente.

##### Flujo:

- Solicitud `DELETE` enviada a `/tareas/:id` (por ejemplo, `/tareas/1`).
- El controlador pasa el `id` de la tarea al servicio, que la elimina.
- La vista envía un mensaje de éxito confirmando la eliminación.

#### Resumen general de las funciones:

Cada una de estas funciones sigue el patrón MVC, donde el **controlador** actúa como intermediario entre las solicitudes del cliente (por ejemplo, a través de Postman) y las respuestas generadas por la aplicación. El controlador:

- **Recibe la solicitud** desde el cliente.
- **Llama a la capa de servicio**, que maneja la lógica de negocio.
- **Envía la respuesta** formateada, ya sea una lista de tareas o un mensaje de éxito, utilizando la capa de vista.

---

#### 4. Capa de Servicio (Service)

La **Capa de Servicio** se encarga de contener la **lógica de negocio**. Es el intermediario entre el controlador y la capa de persistencia. La capa de servicio interactúa con el modelo y la persistencia para filtrar, validar o

manipular los datos antes de que se los pase al controlador.

### Ejemplo Teórico

#### Archivo: `tareaService.mjs`

```
// Importa la capa de persistencia (repositorio)
import TareaRepository from '../repository/tareaRepository.mjs';
import Tarea from '../models/tarea.mjs'; // Importa el modelo de Tarea

// Instancia el repositorio para manejar las tareas
const tareaRepo = new TareaRepository();

// Servicio para obtener todas las tareas
export function listarTareas() {
    // Obtiene todas las tareas desde la capa de persistencia
    return tareaRepo.obtenerTodas();
}

// Servicio para obtener solo las tareas completadas
export function listarTareasCompletadas() {
    // Obtiene todas las tareas desde la capa de persistencia
    const tareas = tareaRepo.obtenerTodas();
    // Filtra las tareas completadas
    return tareas.filter(tarea => tarea.completado);
}

// Servicio para crear una nueva tarea
export function crearTarea(id, titulo, descripcion, completado = false) {
    // Obtiene todas las tareas
    const tareas = tareaRepo.obtenerTodas();
    // Crea una nueva instancia del modelo Tarea
    const nuevaTarea = new Tarea(id, titulo, descripcion, completado);
    // Valida que la tarea tenga un título válido
    nuevaTarea.validar();
    // Añade la nueva tarea a la lista de tareas
    tareas.push(nuevaTarea);
    // Guarda la lista actualizada de tareas en el archivo
    tareaRepo.guardar(tareas);
}

// Servicio para marcar una tarea como completada
export function completarTarea(id) {
    // Obtiene todas las tareas
    const tareas = tareaRepo.obtenerTodas();
    // Encuentra la tarea por ID
    const tarea = tareas.find(tarea => tarea.id === id);
    // Si la tarea existe, la marca como completada
    if (tarea) {
        tarea.completar();
        // Guarda los cambios en el archivo
        tareaRepo.guardar(tareas);
    }
}
```

```
    }

// Servicio para eliminar una tarea
export function eliminarTarea(id) {
    // Obtiene todas las tareas
    let tareas = tareaRepo.obtenerTodas();
    // Elimina la tarea que coincide con el ID
    tareas = tareas.filter(tarea => tarea.id !== id);
    // Guarda la lista actualizada de tareas
    tareaRepo.guardar(tareas);
}
```

## Explicación teórica de cada función en la capa de servicio

### 1. listarTareas()

**Función:** Servicio para obtener todas las tareas.

#### Explicación teórica:

Este servicio simplemente se encarga de llamar a la capa de persistencia (el repositorio) para obtener la lista completa de tareas almacenadas. No realiza ninguna modificación o procesamiento adicional sobre los datos.

#### Flujo:

- Llama al repositorio para obtener las tareas.
  - Devuelve la lista de tareas tal cual fue obtenida.
- 

### 2. listarTareasCompletadas()

**Función:** Servicio para obtener solo las tareas completadas.

#### Explicación teórica:

Este servicio es responsable de filtrar las tareas que están marcadas como completadas. Primero, obtiene todas las tareas desde la capa de persistencia, y luego aplica un filtro para devolver únicamente aquellas cuyo estado es `completado`.

#### Flujo:

- Obtiene todas las tareas desde el repositorio.
  - Filtra las tareas que están marcadas como completadas.
  - Devuelve la lista filtrada.
- 

### 3. crearTarea(id, titulo, descripcion, completado = false)

**Función:** Servicio para crear una nueva tarea.

**Explicación teórica:**

Este servicio es responsable de agregar una nueva tarea. Recibe los datos de la nueva tarea desde el controlador y realiza las siguientes acciones:

- Crea una nueva instancia del modelo `Tarea`.
- Valida que la tarea tenga un título (regla de negocio).
- Añade la tarea a la lista de tareas existentes.
- Finalmente, guarda la lista actualizada de tareas en la persistencia.

**Flujo:**

- Crea una nueva tarea usando el modelo.
- Valida que el título de la tarea no esté vacío.
- Añade la tarea a la lista existente.
- Guarda las tareas actualizadas en la persistencia.

---

**4. `completarTarea(id)`**

**Función:** Servicio para marcar una tarea como completada.

**Explicación teórica:**

Este servicio localiza una tarea en particular por su `id` y la marca como completada. Llama al repositorio para obtener todas las tareas, encuentra la tarea con el `id` proporcionado y la actualiza. Luego guarda los cambios en la capa de persistencia.

**Flujo:**

- Busca la tarea por su `id` en la lista.
- Marca la tarea como completada.
- Guarda los cambios en la persistencia.

---

**5. `eliminarTarea(id)`**

**Función:** Servicio para eliminar una tarea.

**Explicación teórica:**

Este servicio se encarga de eliminar una tarea específica. Filtra la lista de tareas para eliminar la que coincide con el `id` proporcionado. Luego, guarda la lista actualizada de tareas en la persistencia.

**Flujo:**

- Filtra las tareas para eliminar la tarea con el `id` correspondiente.

- Guarda la lista de tareas actualizada en la persistencia.
- 

## Resumen general de la capa de servicio

En el contexto del patrón MVC:

- **La capa de servicio** es responsable de la lógica de negocio. Aquí es donde se toman decisiones, como validar los datos, filtrar listas, o aplicar reglas de negocio (por ejemplo, asegurarse de que una tarea tenga un título antes de ser creada).
  - **La capa de servicio** no interactúa directamente con los detalles de cómo se almacenan los datos; en su lugar, delega esas responsabilidades a la capa de persistencia (repositorio), lo que permite cambiar la fuente de almacenamiento en el futuro sin afectar la lógica de negocio.
  - **El servicio** recibe solicitudes del controlador, ejecuta la lógica de negocio necesaria y luego devuelve los resultados al controlador para ser enviados como respuesta al cliente.
- 

## 5. Capa de Persistencia (Repository)

La capa de persistencia se encarga de interactuar con el sistema de almacenamiento de datos, ya sea una base de datos o un archivo de texto. En este caso, vamos a implementar una **interfaz de persistencia** que permita desacoplar el sistema de almacenamiento (por ejemplo, en un futuro, si queremos cambiar de un archivo de texto a una base de datos). Este diseño garantiza que la lógica de la aplicación no dependa de cómo o dónde se almacenan los datos, permitiendo un **cambio de base de datos** sin modificar el resto del sistema.

### Justificación Teórica

El diseño de la capa de persistencia con una **interfaz** sigue el principio de **desacoplamiento** y **responsabilidad única** del diseño de software. Esto permite que las otras capas (como la de servicios o controladores) no tengan que conocer los detalles de cómo se gestionan los datos a nivel de persistencia. El **desacoplamiento** facilita la **mantenibilidad** y **extensibilidad** del sistema, ya que si en el futuro se cambia la base de datos (por ejemplo, de archivos a una base de datos relacional), solo se tendrá que modificar o implementar una nueva clase de persistencia, sin afectar a la lógica de negocio o de presentación.

### Diagrama Conceptual de la Capa de Persistencia

1. **Interfaz de Persistencia ( `TareasDataSource.mjs` )**: Define los métodos que cualquier implementación de persistencia debe seguir, asegurando que cualquier fuente de datos (archivos, base de datos, etc.) mantenga la misma interfaz de comunicación.
  2. **Implementación Concreta ( `TareaRepository.mjs` )**: Implementa la interfaz para manejar la persistencia usando un archivo de texto. En un futuro, podríamos crear otra clase (por ejemplo, `TareaDatabaseRepository`) que implemente esta misma interfaz pero utilizando una base de datos.
-

## Archivo 1: TareasDataSource.mjs (Interfaz de Persistencia)

Este archivo define una interfaz de persistencia que cualquier implementación concreta debe seguir. Esto asegura que las funciones de persistencia sigan un **contrato** común, permitiendo cambiar el sistema de almacenamiento sin afectar al resto de la aplicación.

```
// Definimos una clase abstracta que actúa como interfaz para la persistencia de datos
export default class TareasDataSource {

    // Método abstracto para obtener todas las tareas
    obtenerTodas() {
        throw new Error('Este método debe ser implementado por la subclase');
    }

    // Método abstracto para guardar todas las tareas
    guardar(tareas) {
        throw new Error('Este método debe ser implementado por la subclase');
    }

    // Método abstracto para eliminar una tarea por su ID
    eliminar(id) {
        throw new Error('Este método debe ser implementado por la subclase');
    }
}
```

### Explicación Teórica de la Interfaz:

- **Clase abstracta TareasDataSource :** Es una clase abstracta que define los métodos clave (`obtenerTodas()`, `guardar()`, `eliminar()`) que cualquier clase de persistencia concreta debe implementar. La ventaja de este diseño es que el contrato (la forma en que los datos se solicitan y guardan) permanece constante independientemente del sistema de almacenamiento.
- **Métodos abstractos:**
  - `obtenerTodas()` : Este método deberá devolver todas las tareas almacenadas, sin importar si están en una base de datos o en un archivo.
  - `guardar()` : Se encargará de guardar todas las tareas en el sistema de almacenamiento.
  - `eliminar()` : Este método eliminará una tarea específica por su `id`.

---

## Archivo 2: TareaRepository.mjs (Implementación de Persistencia usando Archivos de Texto)

Este archivo contiene la implementación concreta que utiliza archivos de texto (`tareas.txt`) para almacenar y recuperar las tareas. Esta clase extiende la interfaz `TareasDataSource`, lo que asegura que cumple con el contrato definido en la interfaz.

```
import fs from 'fs'; // Importamos el módulo del sistema de archivos de Node.js
import path from 'path'; // Módulo para manejar rutas de archivos
```

```
import { fileURLToPath } from 'url'; // Para obtener la ruta del archivo actual
// Importamos la interfaz de persistencia
import TareasDataSource from './TareasDataSource.mjs';
import Tarea from '../models/tarea.mjs'; // Importamos el modelo Tarea

// Obtener la ruta del archivo de tareas
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
const filePath = path.join(__dirname, '../tareas.txt');

// Implementación concreta que extiende la interfaz TareasDataSource
export default class TareaRepository extends TareasDataSource {
    constructor() {
        super(); // Llamada al constructor de la clase base
    }

    // Implementación del método obtenerTodas()
    obtenerTodas() {
        try {
            // Leer el archivo de texto en formato UTF-8
            const data = fs.readFileSync(filePath, 'utf-8');
            // Convertir el contenido del archivo en un array de objetos JSON
            const tareas = JSON.parse(data);
            // Convertir cada tarea en una instancia de la clase Tarea
            return tareas.map(tareaData => new Tarea(
                tareaData.id,
                tareaData.titulo,
                tareaData.descripcion,
                tareaData.completado
            ));
        } catch (error) {
            // Si ocurre un error, como que el archivo no exista, devolvemos un array vacío
            console.error('Error al leer el archivo de tareas:', error);
            return [];
        }
    }

    // Implementación del método guardar()
    guardar(tareas) {
        try {
            // Convertimos el array de tareas a una cadena JSON con indentación de 2 espacios
            const data = JSON.stringify(tareas, null, 2);
            // Guardar la cadena JSON en el archivo de texto
            fs.writeFileSync(filePath, data, 'utf-8');
        } catch (error) {
            // Si ocurre un error al guardar los datos, mostramos el error
            console.error('Error al guardar las tareas:', error);
        }
    }

    // Implementación del método eliminar()
    eliminar(id) {
        try {
```

```

        const tareas = this.obtenerTodas(); // Obtener todas las tareas existentes
        // Filtrar la tarea por ID
        const tareasActualizadas = tareas.filter(tarea => tarea.id !== id);
        this.guardar(tareasActualizadas); // Guardar la lista actualizada
    } catch (error) {
        console.error('Error al eliminar la tarea:', error);
    }
}
}

```

## Explicación del Código:

### 1. Clase `TareaRepository`:

- **Extiende la interfaz `TareasDataSource`**: Implementa los métodos abstractos definidos en la interfaz, lo que asegura que la clase sigue el contrato para manejar las tareas.
- **Manejo de archivos**: Utiliza el módulo `fs` de Node.js para leer y escribir en el archivo `tareas.txt`.

### 2. Método `obtenerTodas()`:

- Primero, deserializamos el archivo de texto utilizando `JSON.parse()` para convertirlo en un array de objetos planos.
- Después, recorremos cada objeto deserializado y lo **reconstruimos** utilizando el constructor de la clase `Tarea`. De esta manera, garantizamos que las instancias resultantes no solo contengan los datos, sino que también tengan acceso a los métodos de la clase `Tarea` (como `completar()` y `validar()`).
- El método devuelve un array de objetos `Tarea`.

### 3. Método `guardar()`:

- **Función**: Convierte el array de tareas en una cadena JSON usando `JSON.stringify()` y guarda los datos en el archivo `tareas.txt`.
- **Manejo de errores**: Si ocurre un error durante la escritura, este se captura y se muestra en la consola.

### 4. Método `eliminar(id)`:

- **Función**: Filtra el array de tareas para eliminar la tarea que coincide con el `id` proporcionado. Luego, guarda el array actualizado en el archivo.
- **Manejo de errores**: Captura cualquier error al eliminar una tarea y lo muestra en la consola.

## Justificación del Diseño

### 1. Desacoplamiento con la Interfaz:

- Al utilizar una **interfaz de persistencia** (`TareasDataSource`), desacoplamos la lógica de negocio de los detalles de cómo se almacenan los datos. Si en el futuro queremos cambiar de un archivo de texto a una base de datos (por ejemplo, PostgreSQL, MySQL o MongoDB), solo necesitamos crear una nueva clase (como `TareaDatabaseRepository`) que implemente la misma interfaz. Esto asegura que la lógica de la aplicación permanezca inalterada.

## 2. Principio de Responsabilidad Única (SRP):

- La clase `TareaRepository` tiene una responsabilidad única: manejar la persistencia de las tareas utilizando archivos de texto. Esto sigue el **Principio de Responsabilidad Única** (Single Responsibility Principle), uno de los principios fundamentales del diseño de software. Al delegar esta responsabilidad en una clase independiente, el código se vuelve más modular y fácil de mantener.

## 3. Escalabilidad:

- Si en el futuro decides cambiar la base de datos, solo necesitas implementar una nueva clase que extienda la interfaz `TareasDataSource`. Esto hace que el sistema sea escalable y flexible sin afectar el código existente en la capa de servicios o controladores.

## 4. Facilidad de mantenimiento:

- Gracias a la abstracción proporcionada por la interfaz, podemos actualizar o cambiar la implementación de la persistencia sin alterar la lógica de la aplicación, lo que reduce el riesgo de errores y hace que el sistema sea más

fácil de mantener.

---

## Conclusión

Esta capa de persistencia es altamente **modular** y **flexible**, gracias a la interfaz `TareasDataSource`, que define el contrato que cualquier fuente de persistencia debe seguir. El uso de esta interfaz asegura que la aplicación pueda cambiar de sistema de almacenamiento (por ejemplo, de un archivo de texto a una base de datos) sin afectar a la lógica de negocio o de presentación. Además, la implementación concreta que usa archivos de texto (`TareaRepository`) es fácil de mantener y permite un manejo robusto de errores, lo que hace que el sistema sea más tolerante a fallos.

---

## 6. Levantando el servidor: `server.mjs`

El **archivo del servidor** en una aplicación Node.js es el núcleo que maneja las **peticiones HTTP**, **enruta las solicitudes** a los controladores correctos y coordina el flujo de la aplicación. En nuestro caso, estamos utilizando **Express.js**, un marco minimalista para manejar rutas, solicitudes y respuestas de manera eficiente.

Este archivo es fundamental porque:

1. **Inicializa el servidor:** Se encarga de poner en marcha el servidor y comenzar a escuchar las peticiones en un puerto específico.
2. **Define las rutas:** Determina qué acción realizar según la solicitud del cliente (por ejemplo, qué controlador debe manejar una solicitud `GET` o `POST`).
3. **Controla la respuesta al cliente:** En función de las acciones ejecutadas, envía una respuesta al cliente (por ejemplo, devolviendo una lista de tareas en formato JSON).

## Justificación

El archivo del servidor es crucial en el patrón **MVC (Modelo-Vista-Controlador)** porque actúa como el punto de entrada principal para las solicitudes entrantes. Desde aquí, las solicitudes se enrutan a los **controladores** que manejan la lógica de negocio a través de los **servicios y repositorios**. Este diseño mantiene el servidor como una parte independiente que no se preocupa por la lógica de negocio o los detalles de almacenamiento, permitiendo que el servidor sea ligero y fácil de mantener.

---

## Código del Archivo del Servidor (`server.mjs`)

```
import express from 'express'; // Importamos el framework Express
// Importamos los controladores
import { listarTareasController,
         listarTareasCompletadasController,
         crearTareaController,
         completarTareaController,
         eliminarTareaController } from './controllers/tareaController.mjs';

const app = express(); // Inicializamos una aplicación de Express
const PORT = 3000; // Definimos el puerto en el que escuchará el servidor

// Middleware para permitir el manejo de solicitudes con cuerpo en formato JSON
app.use(express.json());

// Rutas
    // Ruta para obtener todas las tareas
app.get('/tareas', listarTareasController);
    // Ruta para obtener las tareas completadas
app.get('/tareas/completadas', listarTareasCompletadasController);
    // Ruta para crear una nueva tarea
app.post('/tareas', crearTareaController);
    // Ruta para marcar una tarea como completada
app.put('/tareas/:id/completar', completarTareaController);
    // Ruta para eliminar una tarea
app.delete('/tareas/:id', eliminarTareaController);

// Iniciar el servidor
app.listen(PORT, () => {
```

```
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

## Explicación del Código del Servidor

### 1. Importación de Módulos:

- **express** : El framework Express se utiliza para crear y gestionar el servidor. Express permite manejar las rutas y solicitudes HTTP de manera muy eficiente.
- **Controladores**: Importamos los controladores desde el archivo `tareaController.mjs`. Estos controladores son responsables de manejar la lógica de cada tipo de solicitud (GET, POST, PUT, DELETE) en función de la ruta.

### 2. Inicialización del Servidor:

- **app = express()** : Creamos una instancia de la aplicación Express. Esto es el punto de entrada para todas las solicitudes HTTP.
- **PORT = 3000** : Definimos el puerto en el que nuestro servidor escuchará las solicitudes. En este caso, el puerto es `3000`.

### 3. Middleware `express.json()`:

- **Función**: Este middleware de Express permite que la aplicación pueda recibir y procesar el cuerpo de las solicitudes en formato JSON. Es crucial para manejar datos de tipo `POST` o `PUT`, donde el cliente envía un cuerpo de datos en JSON (por ejemplo, los datos de una tarea nueva).

### 4. Definición de las Rutas:

- **app.get('/tareas', listarTareasController)** : Define la ruta para obtener todas las tareas utilizando el método `GET`. Cuando se accede a `/tareas`, Express invoca el `listarTareasController`.
- **app.get('/tareas/completadas', listarTareasCompletadasController)** : Similar a la ruta anterior, pero solo devuelve las tareas que han sido completadas.
- **app.post('/tareas', crearTareaController)** : Define la ruta para crear una nueva tarea utilizando el método `POST`. El controlador `crearTareaController` es invocado para manejar la lógica de creación.
- **app.put('/tareas/:id/completar', completarTareaController)** : Define la ruta para marcar una tarea como completada, utilizando el método `PUT`. El parámetro `:id` en la URL se utiliza para identificar qué tarea debe completarse.
- **app.delete('/tareas/:id', eliminarTareaController)** : Define la ruta para eliminar una tarea específica usando el método `DELETE`. El `id` se pasa como parámetro para identificar la tarea a eliminar.

### 5. Iniciar el Servidor:

- `app.listen(PORT)` : Esta función inicia el servidor en el puerto especificado. En este caso, el servidor escuchará en el puerto `3000`.
  - **Callback:** El callback asociado imprime un mensaje en la consola, indicando que el servidor está funcionando correctamente y listo para aceptar solicitudes.
- 

## Justificación del Diseño

### 1. Modularidad:

- La lógica de negocio no se mezcla con la lógica de enrutamiento en este archivo. En su lugar, los controladores se encargan de manejar las solicitudes y realizar las operaciones correspondientes. Esto sigue el **principio de responsabilidad única**, asegurando que cada parte del sistema tiene su propia responsabilidad.
- Este diseño también asegura que el servidor sea **modular** y **escalable**. Si se necesitan agregar más rutas o modificar alguna existente, se puede hacer sin tocar el resto de la aplicación.

### 2. Facilidad de Mantenimiento:

- El uso de **controladores** permite que el código sea más fácil de mantener. Si se necesita cambiar la lógica de cómo se obtiene o procesa una tarea, solo se modifica el controlador correspondiente, sin necesidad de tocar el archivo del servidor.
- El hecho de que cada ruta esté claramente definida también facilita la **mantenibilidad** del código. Las solicitudes y respuestas están bien estructuradas, y cada ruta tiene un controlador bien definido.

### 3. Escalabilidad:

- Dado que las rutas están bien organizadas y separadas de la lógica de negocio, el sistema puede escalar fácilmente. Se pueden añadir más rutas o modificar el comportamiento de las rutas existentes sin afectar la arquitectura general del sistema.

En resumen, este archivo del servidor actúa como el punto de entrada a nuestra aplicación, gestionando las solicitudes y delegando la lógica de negocio a los controladores. Este diseño asegura que el sistema sea modular, fácil de mantener y escalable.

---

## Pruebas en Postman para entender el patrón MVC

### 1. Obtener todas las tareas (GET)

**Justificación:** Esta prueba permite a los estudiantes visualizar cómo la aplicación MVC maneja una solicitud para obtener los datos almacenados en el archivo `tareas.txt`. En este caso, el controlador interactúa con el servicio, que a su vez llama a la capa de persistencia para devolver todas las tareas.

- **Método HTTP:** GET
- **URL:** `http://localhost:3000/tareas`
- **Descripción:** Devuelve una lista de todas las tareas.
- **Respuesta esperada:**

```
[  
  { "id": 1, "titulo": "Tarea 1", "descripcion": "Descripción de tarea 1", "completa": false  
  { "id": 2, "titulo": "Tarea 2", "descripcion": "Descripción de tarea 2", "completa": false  
  { "id": 3, "titulo": "Tarea 3", "descripcion": "Descripción de tarea 3", "completa": false  
]
```

## 2. Obtener tareas completadas (GET)

**Justificación:** Esta prueba muestra cómo el servicio filtra los datos, ya que solo queremos obtener las tareas que están marcadas como completadas.

- **Método HTTP:** GET
- **URL:** `http://localhost:3000/tareas/completadas`
- **Descripción:** Devuelve solo las tareas completadas.
- **Respuesta esperada:**

```
[  
  { "id": 2, "titulo": "Tarea 2", "descripcion": "Descripción de tarea 2", "completa": true  
  { "id": 3, "titulo": "Tarea 3", "descripcion": "Descripción de tarea 3", "completa": true  
]
```

## 3. Añadir una nueva tarea (POST)

**Justificación:** Con esta prueba, los estudiantes aprenden cómo el MVC maneja la creación de nuevos datos. Aquí, el controlador recibe los datos enviados a través de la solicitud y se los pasa al servicio, que los valida y los guarda en el archivo.

- **Método HTTP:** POST
- **URL:** `http://localhost:3000/tareas`
- **Descripción:** Crea una nueva tarea.
- **Cuerpo (Body):**

```
{  
  "id": 4,  
  "titulo": "Tarea 4",  
  "descripcion": "Descripción de tarea 4",  
  "completado": false  
}
```

- **Respuesta esperada:**

```
{
```

```
        "message": "Tarea creada con éxito"
    }
```

#### 4. Marcar tarea como completada (PUT)

**Justificación:** Esta prueba permite entender cómo actualizar un recurso en el patrón MVC. El controlador recibe la solicitud de actualización, el servicio modifica la tarea y la persistencia actualiza el archivo.

- **Método HTTP:** PUT
- **URL:** `http://localhost:3000/tareas/1/completar`
- **Descripción:** Marca la tarea con `id = 1` como completada.
- **Respuesta esperada:**

```
{
  "message": "Tarea marcada como completada"
}
```

#### 5. Eliminar una tarea (DELETE)

**Justificación:** Finalmente, esta prueba muestra cómo el MVC maneja la eliminación de recursos. El controlador recibe la solicitud de eliminar una tarea, el servicio se asegura de que exista, y el repositorio elimina la tarea del archivo.

- **Método HTTP:** DELETE
- **URL:** `http://localhost:3000/tareas/1`
- **Descripción:** Elimina la tarea con `id = 1`.
- **Respuesta esperada:**

```
{
  "message": "Tarea eliminada con éxito"
}
```

#### Explicación general de las pruebas en Postman:

Estas pruebas enseñan a los estudiantes cómo funciona cada componente del patrón MVC:

- **Modelo:** Representa las tareas.
- **Vista:** Devuelve las tareas en formato JSON.
- **Controlador:** Gestiona las solicitudes y respuestas.
- **Servicio:** Procesa la lógica de negocio (filtrado, validación).
- **Persistencia:** Lee y escribe datos en un archivo de texto.

---

## Conclusión

Con esta implementación basada en **MVC** en Node.js, hemos logrado:

1. Separar claramente las responsabilidades de cada capa (Modelo, Vista, Controlador, Servicio y Persistencia).
2. Utilizar un **nivel de abstracción** en la capa de persistencia para facilitar la transición a otras fuentes de datos en el futuro.
3. Crear una aplicación que lee datos desde un archivo de texto utilizando la API de Node.js (`fs`) y los presenta al usuario en formato JSON.

Este enfoque modular permite que la aplicación sea fácil de mantener y escalar, con la flexibilidad de cambiar la fuente de datos en el futuro sin necesidad de modificar la lógica de negocio.

---

## Indice 3 - Trabajo Práctico 4: Implementación de un Servidor con Express y Arquitectura MVC

En este trabajo práctico, implementaremos un servidor con **Node.js** y **Express** que seguirá la arquitectura **Modelo-Vista-Controlador (MVC)**. El servidor se ejecutará en el puerto **3005** y escuchará diversas peticiones **GET**.

---

### Requerimientos del Trabajo

1. **Levantar un servidor Express en el puerto 3005.**
2. El servidor debe escuchar varias rutas GET:
  - **/superheroes/id/:id:** Recibe un ID de superhéroe y devuelve los datos de ese superhéroe o un mensaje si no fue encontrado.
  - **/superheroes/atributo/:atributo/:valor:** Recibe un atributo (por ejemplo, nombre o poder) y devuelve una lista de superhéroes que cumplen con ese criterio.
  - **/superheroes/edad/mayorA30:** Devuelve una lista de superhéroes mayores de 30 años que además sean del planeta Tierra y tengan al menos 2 poderes.

### Lista de Superhéroes

La lista de superhéroes proviene del archivo `superheroes.txt`:

```
[  
  {  
    "id": 1,  
    "nombreSuperHeroe": "Spiderman",  
    "nombreReal": "Peter Parker",  
    "nombreSociedad": "Vigilante",  
    "edad": 25,
```

```

        "planetaOrigen": "Tierra",
        "debilidad": "Radioactiva",
        "poder": ["Tregar paredes", "Sentido arácnido", "Super fuerza", "Agilidad"],
        "habilidadEspecial": "Redes de telaraña",
        "aliado": ["Ironman"],
        "enemigo": ["Duende Verde"]
    },
    {
        "id": 2,
        "nombreSuperHeroe": "Ironman",
        "nombreReal": "Tony Stark",
        "nombreSociedad": "Vigilante",
        "edad": 45,
        "planetaOrigen": "Tierra",
        "debilidad": "Dependiente de la tecnología",
        "poder": ["Armadura blindada", "Volar", "Láseres"],
        "habilidadEspecial": "Ingeniería avanzada",
        "aliado": ["Spiderman"],
        "enemigo": ["Mandarín"]
    },
    {
        "id": 3,
        "nombreSuperHeroe": "Thor",
        "nombreReal": "Thor Odinson",
        "nombreSociedad": "Dios del Trueno",
        "edad": 1000,
        "planetaOrigen": "Asgard",
        "debilidad": "Destruir su martillo",
        "poder": ["Controlar el trueno", "Fuerza sobrehumana", "Inmortal"],
        "habilidadEspecial": "Invocación de tormentas",
        "aliado": ["Loki"],
        "enemigo": ["Hela"]
    }
]

```

---

## Estructura del Proyecto

```

/project-root
|-- /models
|   |-- superheroe.mjs
|
|-- /controllers
|   |-- superheroesController.mjs
|
|-- /services
|   |-- superheroesService.mjs
|
|-- /repository
|   |-- superheroesRepository.mjs
|   |-- superheroesDataSource.mjs

```

```
|  
|-- /views  
|   |-- responseView.mjs  
|  
|-- server.mjs  
|-- superheroes.txt
```

---

## 1. Capa de Persistencia

### Abstracción de la Persistencia

Este archivo define una abstracción que otras clases de persistencia deben implementar:

```
// repository/superheroesDataSource.mjs  
export default class SuperheroesDataSource {  
    // Método abstracto para obtener todos los superhéroes  
    obtenerTodos() {  
        throw new Error('Este método debe ser implementado por la subclase');  
    }  
}
```

### Implementación de la Persistencia con Archivos

Esta clase implementa el método `obtenerTodos()` que lee los datos desde `superheroes.txt`.

```
// repository/superheroesRepository.mjs  
import fs from 'fs';  
import path from 'path';  
import { fileURLToPath } from 'url';  
import SuperheroesDataSource from './superheroesDataSource.mjs';  
  
const __filename = fileURLToPath(import.meta.url);  
const __dirname = path.dirname(__filename);  
  
export default class SuperheroesFileRepository extends SuperheroesDataSource {  
    constructor() {  
        super();  
        this.filePath = path.join(__dirname, '../superheroes.txt');  
    }  
  
    obtenerTodos() {  
        const data = fs.readFileSync(this.filePath, 'utf-8');  
        return JSON.parse(data); // Convierte el archivo JSON en un array de objetos JS  
    }  
}
```

## 2. Capa de Servicio

La **Capa de Servicio** contiene la lógica de negocio para las diversas solicitudes. Interactúa con la capa de persistencia para obtener los datos.

```
// services/superheroesService.mjs
import SuperheroesRepository from '../repository/superheroesRepository.mjs';

const repository = new SuperheroesRepository();

export function obtenerSuperheroePorId(id) {
  const superheroes = repository.obtenerTodos();
  return superheroes.find(hero => hero.id === id);
}

export function buscarSuperheroesPorAtributo(atributo, valor) {
  const superheroes = repository.obtenerTodos();
  return superheroes.filter(hero =>
    String(hero[atributo]).toLowerCase().includes(valor.toLowerCase())
  );
}

export function obtenerSuperheroesMayoresDe30() {
  const superheroes = repository.obtenerTodos();
  return superheroes.filter(hero =>
    hero.edad > 30 && hero.planetaOrigen === 'Tierra' && hero.poder.length >= 2
  );
}
```

---

## 3. Controlador

El **Controlador** maneja las solicitudes HTTP y utiliza la capa de servicio para obtener los datos necesarios.

```
// controllers/superheroesController.mjs
import { obtenerSuperheroePorId, buscarSuperheroesPorAtributo,
obtenerSuperheroesMayoresDe30 } from '../services/superheroesService.mjs';
import { renderizarSuperheroe, renderizarListaSuperheroes }
from '../views/responseView.mjs';

export function obtenerSuperheroePorIdController(req, res) {
  const { id } = req.params;
  const superhero = obtenerSuperheroePorId(parseInt(id));

  if (superhero) {
    res.send(renderizarSuperheroe(superhero));
  } else {
    res.status(404).send({ mensaje: "Superhéroe no encontrado" });
  }
}
```

```

}

export function buscarSuperheroesPorAtributoController(req, res) {
  const { atributo, valor } = req.params;
  const superheroes = buscarSuperheroesPorAtributo(atributo, valor);

  if (superheroes.length > 0) {
    res.send(renderizarListaSuperheroes(superheroes));
  } else {
    res.status(404).send({ mensaje: "No se encontraron superhéroes con ese atributo" });
  }
}

export function obtenerSuperheroesMayoresDe30Controller(req, res) {
  const superheroes = obtenerSuperheroesMayoresDe30();
  res.send(renderizarListaSuperheroes(superheroes));
}

```

---

## 4. Vista

La **Vista** es responsable de formatear los datos en un formato adecuado, en este caso, **JSON**.

```

// views/responseView.mjs
export function renderizarSuperheroe(superheroe) {
  return JSON.stringify(superheroe, null, 2);
}

export function renderizarListaSuperheroes(superheroes) {
  return JSON.stringify(superheroes, null, 2);
}

```

---

## 5. Servidor Express

El **servidor Express** se configura para escuchar en el puerto **3005** y manejar las solicitudes a las diversas rutas.

```

// server.mjs
import express from 'express';
import { obtenerSuperheroePorIdController, buscarSuperheroesPorAtributoController, obtenerS
from './controllers/superheroesController.mjs';

const app = express();
const PORT = 3005;

// Rutas
app.get('/superheroes/id/:id', obtenerSuperheroePorIdController);

```

```
app.get('/superheroes/atributo/:atributo/:valor', buscarSuperheroesPorAtributoController);
app.get('/superheroes/edad/mayorA30', obtenerSuperheroesMayoresDe30Controller);

// Levantar el servidor en el puerto 3005
app.listen(PORT, () => {
  console.log(`Servidor corriendo en el puerto ${PORT}`);
});
```

---

## Conclusión

Este trabajo práctico implementa un servidor Express con la arquitectura MVC, añadiendo una capa de servicio para manejar la lógica de negocio y una capa de persistencia para interactuar con los datos almacenados en un archivo `superheroes.txt`. La separación de responsabilidades en este patrón facilita la escalabilidad y la mantenibilidad del proyecto, permitiendo realizar cambios en la fuente de datos o la lógica de negocio sin afectar otras partes de la aplicación.