

# Sprint 1 - Trabajo Práctico 4

---

## Fundamentación: Introducción a la Arquitectura MVC con Node.js

La **arquitectura Modelo-Vista-Controlador (MVC)** es uno de los patrones de diseño más comunes y efectivos para el desarrollo de aplicaciones web modernas. En su estructura, divide una aplicación en tres componentes distintos:

1. **Modelo (Model)**: Encargado de la lógica de negocio y la interacción con la base de datos o las fuentes de datos.
2. **Vista (View)**: Responsable de la presentación de los datos al usuario, usualmente en HTML o, en el caso de APIs, en formato JSON.
3. **Controlador (Controller)**: Actúa como intermediario entre la Vista y el Modelo, gestionando las solicitudes del cliente, ejecutando la lógica de negocio a través del Modelo y proporcionando los datos formateados a la Vista.

Este patrón es particularmente útil en **Node.js** debido a su naturaleza asíncrona y su capacidad de manejar múltiples solicitudes simultáneas de manera eficiente. Según **Node.js for Beginners (2024)**, la separación de responsabilidades que proporciona MVC facilita el desarrollo modular, la escalabilidad, el mantenimiento del código y la colaboración en proyectos a gran escala.

Además de las capas principales del MVC, también se suelen añadir capas adicionales en aplicaciones más grandes y modulares:

- **Capa de Servicio (Service)**: Para gestionar la lógica de negocio y actuar como intermediaria entre el Controlador y la Capa de Persistencia.
  - **Capa de Persistencia (Persistence)**: Encargada de la interacción con la fuente de datos. Puede utilizar un nivel de abstracción, lo que permite cambiar fácilmente la fuente de datos sin afectar la lógica del sistema.
- 

## Parte Teórica: Implementación de MVC con Node.js

---

En esta sección teórica, implementaremos el patrón **MVC** en una aplicación que gestiona **tareas**, utilizando los métodos HTTP más comunes: **GET, POST, PUT, y DELETE**. Además, la Capa de Persistencia utilizará archivos de texto como fuente de datos, con un nivel de abstracción que permitirá cambiar la fuente de datos en el futuro sin afectar la lógica de la aplicación.

### Componentes del MVC

## 1. Modelo (Model)

El **Modelo** se encarga de definir cómo se estructuran los datos y qué operaciones se pueden realizar sobre ellos. En este ejemplo, los datos son las **tareas** y el modelo define los atributos y métodos asociados a cada tarea.

```
// models/tarea.mjs
export default class Tarea {
  constructor(id, titulo, descripcion, completado = false) {
    this.id = id;
    this.titulo = titulo;
    this.descripcion = descripcion;
    this.completado = completado;
  }

  completar() {
    this.completado = true;
  }
}
```

## 2. Vista (View)

La **Vista** es responsable de formatear los datos que se devolverán al usuario. En este caso, las respuestas serán en **JSON**.

```
// views/tareaView.mjs
export function renderizarListaTareas(tareas) {
  return JSON.stringify(tareas, null, 2);
}

export function renderizarTarea(tarea) {
  return JSON.stringify(tarea, null, 2);
}
```

## 3. Controlador (Controller)

El **Controlador** es el intermediario entre la Vista y el Modelo. Recibe las solicitudes HTTP, invoca los métodos apropiados del Modelo (o la Capa de Servicio), y devuelve una respuesta adecuada.

```
// controllers/tareaController.mjs
import { obtenerTareas, obtenerTareaPorId, agregarTarea, actualizarTarea, eliminarTarea } from '../services/tareaService.mjs';
import { renderizarListaTareas, renderizarTarea } from '../views/tareaView.mjs';

export function obtenerTodasLasTareas(req, res) {
  const tareas = obtenerTareas();
  res.send(renderizarListaTareas(tareas));
}

export function obtenerTareaPorIdController(req, res) {
  const { id } = req.params;
  const tarea = obtenerTareaPorId(parseInt(id));
  if (tarea) {
```

```

        res.send(renderizarTarea(tarea));
    } else {
        res.status(404).send({ mensaje: "Tarea no encontrada" });
    }
}

export function agregarNuevaTarea(req, res) {
    const datos = req.body;
    const nuevaTarea = agregarTarea(datos);
    res.send(nuevaTarea);
}

export function actualizarTareaPorId(req, res) {
    const { id } = req.params;
    const datos = req.body;
    const resultado = actualizarTarea(parseInt(id), datos);
    res.send(resultado ? 'Tarea actualizada' : 'Tarea no encontrada');
}

export function eliminarTareaPorId(req, res) {
    const { id } = req.params;
    const resultado = eliminarTarea(parseInt(id));
    res.send(resultado ? 'Tarea eliminada' : 'Tarea no encontrada');
}

```

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

La **Capa de Servicio** contiene la lógica de negocio y se comunica con la Capa de Persistencia para gestionar las operaciones relacionadas con las tareas.

```

// services/tareaService.mjs
import TareaRepository from '../persistence/tareaRepository.mjs';

const repository = new TareaRepository();

export function obtenerTareas() {
    return repository.obtenerTodas();
}

export function obtenerTareaPorId(id) {
    const tareas = repository.obtenerTodas();
    return tareas.find(t => t.id === id);
}

export function agregarTarea(datos) {
    const tareas = repository.obtenerTodas();
    const nuevaTarea = { id: tareas.length + 1, ...datos };
    tareas.push(nuevaTarea);
    repository.guardar(tareas);
    return nuevaTarea;
}

export function actualizarTarea(id, datos) {
    const tareas = repository.obtenerTodas();
    const tarea = tareas.find(t => t.id === id);

```

```

    if (tarea) {
      Object.assign(tarea, datos);
      repository.guardar(tareas);
      return true;
    }
    return false;
}

export function eliminarTarea(id) {
  const tareas = repository.obtenerTodas();
  const nuevaLista = tareas.filter(t => t.id !== id);
  if (tareas.length !== nuevaLista.length) {
    repository.guardar(nuevaLista);
    return true;
  }
  return false;
}

```

## 5. Capa de Persistencia (Persistence)

La **Capa de Persistencia** maneja la interacción con la fuente de datos (en este caso, un archivo de texto). Utiliza una abstracción para que la fuente de datos se pueda cambiar fácilmente.

### Abstracción de Persistencia:

```

// persistence/tareaDataSource.mjs
export default class TareaDataSource {
  obtenerTodas() {
    throw new Error("Este método debe ser implementado por la subclase");
  }

  guardar() {
    throw new Error("Este método debe ser implementado por la subclase");
  }
}

```

### Implementación utilizando archivos:

```

// persistence/tareaRepository.mjs
import fs from 'fs';
import path from 'path';
import TareaDataSource from './tareaDataSource.mjs';

export default class TareaFileRepository extends TareaDataSource {
  constructor() {
    super();
    this.filePath = path.join(__dirname, '../tareas.txt');
  }

  obtenerTodas() {
    const data = fs.readFileSync(this.filePath, 'utf-8');
    return JSON.parse(data);
  }
}

```

```

    guardar(tareas) {
      fs.writeFileSync(this.filePath, JSON.stringify(tareas, null, 2));
    }
}

```

## 6. Servidor Express

Finalmente, el servidor Express maneja las solicitudes HTTP y las distribuye a los controladores correspondientes.

```

// server.mjs
import express from 'express';
import { obtenerTodasLasTareas, obtenerTareaPorIdController,
  agregarNuevaTarea, actualizarTareaPorId, eliminarTareaPorId } from './controllers/tareaController.mjs';

const app = express();
app.use(express.json());

const PORT = 3000;

app.get('/tareas', obtenerTodasLasTareas);
app.get('/tareas/:id', obtenerTareaPorIdController);
app.post('/tareas', agregarNuevaTarea);
app.put('/tareas/:id', actualizarTareaPorId);
app.delete('/tareas/:id', eliminarTareaPorId);

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

```

## 7. Archivo de Datos - `tareas.txt`

Este archivo almacenará las tareas en formato JSON. Se debe colocar en la raíz del proyecto.

```

[
  {
    "id": 1,
    "titulo": "Comprar comida",
    "descripcion": "Ir al supermercado para comprar víveres.",
    "completado": false
  },
  {
    "id": 2,
    "titulo": "Estudiar Node.js",
    "descripcion": "Revisar el capítulo sobre Express.",
    "completado": true
  }
]

```

Este archivo servirá como fuente de datos que será manipulada por la **Capa de Persistencia**.

## Estructura del Proyecto

### Archivos del proyecto

Aquí está el árbol de directorios del proyecto completo:

```
📁 /project-root
  └── 📁 /models
    |   └── 📄 tarea.mjs
  └── 📁 /controllers
    |   └── 📄 tareaController.mjs
  └── 📁 /services
    |   └── 📄 tareaService.mjs
  └── 📁 /persistence
    |   ├── 📄 tareaRepository.mjs
    |   └── 📄 tareaDataSource.mjs
  └── 📁 /views
    |   └── 📄 tareaView.mjs
  └── 📄 server.mjs
  └── 📄 tareas.txt
```

## Pruebas con Postman - Parte Teórica

---

### 1. Probar con GET

Obtener todas las tareas:

- **Método:** GET
- **URL:** `http://localhost:3000/tareas`

Obtener una tarea por ID:

- **Método:** GET
- **URL:** `http://localhost:3000/tareas/1`

### 2. Probar con POST

Agregar una nueva tarea:

- **Método:** POST
- **URL:** `http://localhost:3000/tareas`
- **Body (JSON):**

```
{
  "titulo": "Nueva Tarea",
  "descripcion": "Descripción de la nueva tarea",
  "completado": false
}
```

### 3. Probar con PUT

Actualizar una tarea por ID:

- **Método:** PUT
- **URL:** `http://localhost:3000/tareas/1`
- **Body (JSON):**

```
{  
  "titulo": "Tarea Actualizada",  
  "completado": true  
}
```

### 4. Probar con DELETE

Eliminar una tarea por ID:

- **Método:** DELETE
- **URL:** `http://localhost:3000/tareas/2`

---

## Trabajo Práctico 4: Implementación de un Servidor con Node.js usando la Arquitectura MVC

### Objetivo del Trabajo Práctico

En este trabajo práctico, vamos a implementar un servidor en **Node.js** que gestiona información de **superhéroes**, utilizando la arquitectura **Modelo-Vista-Controlador (MVC)**. El objetivo es crear un sistema modular y organizado que permita realizar operaciones de lectura, creación, actualización y eliminación (CRUD) de datos de superhéroes almacenados en un archivo. Además, implementaremos endpoints especiales que permitirán obtener superhéroes filtrados por diferentes criterios, como la edad, el origen o el número de poderes.

El servidor deberá manejar los siguientes métodos HTTP:

- **GET** para recuperar información de superhéroes.
- **POST** para agregar un nuevo superhéroe.
- **PUT** para actualizar un superhéroe existente.
- **DELETE** para eliminar un superhéroe.
- **GET con filtros** para obtener superhéroes según criterios específicos.

Finalmente, probaremos el funcionamiento del servidor usando **Postman**, una herramienta que permite realizar pruebas sobre APIs y servidores.

---

### Requerimientos del Trabajo

1. Levantar un servidor en el puerto **3005**.
  2. Implementar los siguientes endpoints:
    - **GET /superheroes**: Devuelve todos los superhéroes.
    - **GET /superheroes/:id**: Devuelve un superhéroe por su ID.
    - **POST /superheroes**: Agrega un nuevo superhéroe.
    - **PUT /superheroes/:id**: Actualiza un superhéroe por su ID.
    - **DELETE /superheroes/:id**: Elimina un superhéroe por su ID.
    - **GET /superheroes/atributo/:atributo/:valor**: Devuelve una lista de superhéroes que cumplen con un atributo específico (nombre, edad, planeta, etc.).
    - **GET /superheroes/filtros/avanzados**: Devuelve superhéroes mayores de 30 años, que sean de la Tierra y que tengan al menos 2 poderes.
- 

## 1. Capa de Persistencia

La **Capa de Persistencia** se encarga de manejar la lectura y escritura de datos en el archivo que simula una base de datos.

### Abstracción de la Persistencia

Este archivo define una interfaz que será implementada por la clase de persistencia.

```
// persistence/superheroesDataSource.mjs
export default class SuperheroesDataSource {
    obtenerTodos() {
        throw new Error("Este método debe ser implementado por la subclase");
    }

    guardar() {
        throw new Error("Este método debe ser implementado por la subclase");
    }
}
```

### Implementación usando archivos

Aquí se implementa la lectura y escritura en el archivo `superheroes.txt`.

```
// persistence/superheroesRepository.mjs
import fs from 'fs';
import path from 'path';
import SuperheroesDataSource from './superheroesDataSource.mjs';

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);
    }

    guardar(superhero) {
        const heroes = this.obtenerTodos();
        heroes.push(superhero);
        fs.writeFileSync(this.filePath, JSON.stringify(heroes));
    }
}
```

```

        return JSON.parse(data);
    }

guardar(superheroes) {
    fs.writeFileSync(this.filePath, JSON.stringify(superheroes, null, 2));
}
}

```

## 2. Capa de Servicio

La **Capa de Servicio** es responsable de gestionar la lógica de negocio y de comunicarse con la **Capa de Persistencia** para obtener, agregar, actualizar o eliminar los datos de los superhéroes.

```

// services/superheroesService.mjs
import SuperheroesRepository from '../persistence/superheroesRepository.mjs';

const repository = new SuperheroesRepository();

export function obtenerSuperheroes() {
    return repository.obtenerTodos();
}

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

export function agregarSuperheroe(datos) {
    const superheroes = repository.obtenerTodos();
    const nuevoSuperheroe = { id: superheroes.length + 1, ...datos };
    superheroes.push(nuevoSuperheroe);
    repository.guardar(superheroes);
    return nuevoSuperheroe;
}

export function actualizarSuperheroe(id, datos) {
    const superheroes = repository.obtenerTodos();
    const superhero = superheroes.find(hero => hero.id === id);
    if (superhero) {
        Object.assign(superhero, datos);
        repository.guardar(superheroes);
        return true;
    }
    return false;
}

export function eliminarSuperheroe(id) {
    const superheroes = repository.obtenerTodos();
    const nuevaLista = superheroes.filter(hero => hero.id !== id);
    if (superheroes.length !== nuevaLista.length) {
        repository.guardar(nuevaLista);
        return true;
    }
}

```

```

        return false;
    }

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

export function obtenerSuperheroesMayoresDe30YConFiltros() {
    const superheroes = repository.obtenerTodos();
    return superheroes.filter(hero => hero.edad > 30 && hero.planetaOrigen === 'Tierra'
        && hero.poderes.length >= 2);
}

```

### 3. Controlador

El **Controlador** recibe las solicitudes del cliente, gestiona la lógica de las operaciones solicitadas mediante la **Capa de Servicio**, y utiliza la **Vista** para devolver las respuestas al cliente.

```

// controllers/superheroesController.mjs
import { obtenerSuperheroes, obtenerSuperheroePorId, agregarSuperheroe
    , actualizarSuperheroe, eliminarSuperheroe, buscarSuperheroesPorAtributo
    , obtenerSuperheroesMayoresDe30YConFiltros }
    from '../services/superheroesService.mjs';
import { renderizarSuperheroe, renderizarListaSuperheroes }
    from '../views/responseView.mjs';

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

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 agregarNuevoSuperheroe(req, res) {
    const datos = req.body;
    const nuevoSuperheroe = agregarSuperheroe(datos);
    res.send(nuevoSuperheroe);
}

export function actualizarSuperheroePorId(req, res) {
    const { id } = req.params;
    const datos = req.body;
    const resultado = actualizarSuperheroe(parseInt(id), datos);

```

```

        res.send(resultado ? 'Superhéroe actualizado' : 'Superhéroe no encontrado');
    }

export function eliminarSuperheroePorId(req, res) {
    const { id } = req.params;
    const resultado = eliminarSuperheroe(parseInt(id));
    res.send(resultado ? 'Superhéroe eliminado' : 'Superhéroe no encontrado');
}

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

export function obtenerSuperheroesMayoresDe30YConFiltrosController(req, res) {
    const superheroes = obtenerSuperheroesMayoresDe30YConFiltros();
    res.send(renderizarListaSuperheroes(superheroes));
}

```

---

## 4. Vista

La **Vista** es responsable de formatear los datos en **JSON** para devolverlos al cliente.

```

// 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** es el que maneja las rutas y las solicitudes HTTP del cliente. Este servidor escuchará en el puerto **3005**.

```

// server.mjs
import express from 'express';
import { obtenerTodosLosSuperheroes, obtenerSuperheroePorIdController,
agregarNuevoSuperheroe, actualizarSuperheroePorId, eliminarSuperheroePorId,
buscarSuperheroesPorAtributoController,
obtenerSuperheroesMayoresDe30YConFiltrosController }
from './controllers/superheroesController.mjs';

const app = express();
app.use(express.json());

const PORT = 3005;

```

```

// Rutas
app.get('/superheroes', obtenerTodosLosSuperheroes);
app.get('/superheroes/id/:id', obtenerSuperheroePorIdController);
app.post('/superheroes', agregarNuevoSuperheroe);
app.put('/superheroes/:id', actualizarSuperheroePorId);
app.delete('/superheroes/:id', eliminarSuperheroePorId);
app.get('/superheroes/atributo/:atributo/:valor', buscarSuperheroesPorAtributoController);
app.get('/superheroes/filtros/avanzados',
    obtenerSuperheroesMayoresDe30YConFiltrosController);

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

```

## 6. Modelo (Model) - superhero.mjs

El modelo define cómo se estructuran los datos de los superhéroes y las operaciones que se pueden realizar sobre ellos.

```

// models/superheroe.mjs
export default class Superheroe {
  constructor(id, nombreSuperHeroe, nombreReal, edad, planetaOrigen, debilidad,
    poderes = [], aliados = [], enemigos = []) {

    this.id = id;
    this.nombreSuperHeroe = nombreSuperHeroe;
    this.nombreReal = nombreReal;
    this.edad = edad;
    this.planetaOrigen = planetaOrigen;
    this.debilidad = debilidad;
    this.poderes = poderes;
    this.aliados = aliados;
    this.enemigos = enemigos;
  }

  // Método para agregar un nuevo poder al superhéroe
  agregarPoder(poder) {
    this.poderes.push(poder);
  }

  // Método para agregar un aliado
  agregarAliado(aliado) {
    this.aliados.push(aliado);
  }

  // Método para agregar un enemigo
  agregarEnemigo(enemigo) {
    this.enemigos.push(enemigo);
  }
}

```

Este modelo establece la estructura de un superhéroe y algunos métodos que permiten añadir poderes, aliados o enemigos.

---

## 7. Archivo de Datos - superheroes.txt

---

El archivo `superheroes.txt` contendrá una lista de superhéroes en formato JSON y se colocará en la raíz del proyecto.

En un mundo ideal usan el que generaron al terminar el práctico 3, este se usa para mostrar una resolución de ejemplo.

```
[  
  {  
    "id": 1,  
    "nombreSuperHeroe": "Spiderman",  
    "nombreReal": "Peter Parker",  
    "edad": 25,  
    "planetaOrigen": "Tierra",  
    "debilidad": "Radioactiva",  
    "poderes": ["Tregar paredes", "Sentido arácnido", "Super fuerza", "Agilidad"],  
    "aliados": ["Ironman"],  
    "enemigos": ["Duende Verde"]  
  },  
  {  
    "id": 2,  
    "nombreSuperHeroe": "Ironman",  
    "nombreReal": "Tony Stark",  
    "edad": 45,  
    "planetaOrigen": "Tierra",  
    "debilidad": "Dependiente de la tecnología",  
    "poderes": ["Armadura blindada", "Volar", "Láseres"],  
    "aliados": ["Spiderman"],  
    "enemigos": ["Mandarín"]  
  }  
]
```

Este archivo simulará una base de datos para los superhéroes y será manipulado por la **Capa de Persistencia**.

---

## Estructura del Proyecto

### Archivos del proyecto

```
/project-root  
└── /models  
    └── superhero.mjs  
└── /controllers  
    └── superheroesController.mjs
```

```
└── └── /services
|   └── └── superheroesModule.mjs
└── └── /persistence
|   └── └── superheroesRepository.mjs
|   └── └── superheroesDataSource.mjs
└── └── /views
|   └── └── responseView.mjs
└── └── server.mjs
└── └── superheroes.txt
```

---

## Pruebas con Postman

---

### 1. Probar con GET

Obtener todos los superhéroes:

- **Método:** GET
- **URL:** `http://localhost:3005/superheroes`

Obtener un superhéroe por ID:

- **Método:** GET
- **URL:** `http://localhost:3005/superheroes/id/1`

### 2. Probar con POST

Agregar un nuevo superhéroe:

- **Método:** POST
- **URL:** `http://localhost:3005/superheroes`
- **Body (JSON):**

```
{
  "nombreSuperHeroe": "Hulk",
  "nombreReal": "Bruce Banner",
  "edad": 40,
  "planetaOrigen": "Tierra",
  "debilidad": "Control de emociones",
  "poderes": ["Super fuerza", "Regeneración"],
  "aliados": ["Thor", "Ironman"],
  "enemigos": ["Abominación"]
}
```

### 3. Probar con PUT

Actualizar un superhéroe por ID:

- **Método:** PUT
- **URL:** `http://localhost:3005/superheroes/1`
- **Body (JSON):**

```
{  
  "nombreSuperHeroe": "Spiderman",  
  "edad": 26  
}
```

## 4. Probar con DELETE

### Eliminar un superhéroe por ID:

- **Método:** DELETE
- **URL:** `http://localhost:3005/superheroes/2`

## 5. Probar con Filtros

### Buscar superhéroes por atributo:

- **Método:** GET
- **URL:** `http://localhost:3005/superheroes/atributo/nombreSuperHeroe/Ironman`

### Obtener superhéroes mayores de 30 años, de la Tierra, con más de 2 poderes:

- **Método:** GET
- **URL:** `http://localhost:3005/superheroes/filtros/avanzados`

---

## Conclusión

En este trabajo práctico, hemos implementado un servidor en **Node.js** utilizando la arquitectura **Modelo-Vista-Controlador (MVC)**. El servidor incluye funcionalidades **CRUD** para gestionar superhéroes, así como endpoints avanzados que permiten filtrar datos de manera específica. Además, hemos utilizado **Postman** para probar y verificar cada endpoint.