Ejercitación Práctica: "Boxes & Pitstops" - Gestión de Pilotos de F1

Materia: Desarrollo Web **Docente:** Ing. Alejandro Rey Corvalán

Objetivos:

- Comprender el uso de `express.Router` para organizar rutas API.
- Implementar y entender el propósito de diferentes tipos de middleware:
    - Middleware para manejo de CORS.
    - Middleware para logging de peticiones.
    - Middleware para limpieza básica de datos de entrada.
    - Middleware centralizado para manejo de errores.
- Integrar backend (Node.js, Express, Sequelize) con frontend (HTML5, Bootstrap, Vanilla JS).
- Practicar el desarrollo de una aplicación web completa (cliente-servidor) sin frameworks frontend.

Herramientas:

- Node.js
- npm (o yarn)
- Express
- Sequelize (con SQLite para simplificar)
- HTML5
- Bootstrap (CDN)
- JavaScript (Vanilla JS)
- Visual Studio Code (Editor de Código)

Vamos a construir una pequeña aplicación web llamada **"Boxes & Pitstops"** que nos permitirá gestionar una lista básica de pilotos de F1. Lo importante aquí no es la complejidad de la app, sino que **integremos y entendamos cómo funcionan los diferentes tipos de middleware** y cómo se comunica nuestro backend con un frontend sencillo hecho con HTML, Bootstrap y Vanilla JS.

Paso 0: Preparación del Entorno

1. **Abran Visual Studio Code.**
2. **Creen una nueva carpeta** para el proyecto. Pueden llamarla `f1-pilotos-app`.
3. **Abran la terminal integrada** en VS Code (`Ctrl + Ñ` o `View > Terminal`).
4. **Inicialicen el proyecto de Node.js:**

npm init -y

Esto creará el archivo `package.json`.

5. **Instalen las dependencias necesarias:**

   npm install express sequelize sqlite3 cors

   - `express`: El framework web.
   - `sequelize`: ORM para interactuar con la base de datos.
   - `sqlite3`: Un adaptador para usar Sequelize con SQLite (base de datos simple basada en archivo, ideal para desarrollo y ejercicios).
   - `cors`: Middleware estándar para manejar CORS.

Paso 1: Configuración Básica del Servidor Express

Vamos a crear el archivo principal de nuestra aplicación y poner en marcha un servidor básico.

1. **Creen un archivo** llamado `app.js` en la raíz del proyecto.
2. **Peguen el siguiente código:**

```
// app.js

const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000; // Usamos el puerto 3000 por defecto

// Middleware para parsear JSON en el cuerpo de las peticiones POST/PUT

app.use(express.json());

// Middleware para parsear datos de formularios (si usáramos form-urlencoded)

app.use(express.urlencoded({ extended: true }));

// TODO: Aquí irán nuestros middleware personalizados

// TODO: Aquí irán nuestras rutas API
```

```
// TODO: Aquí serviremos archivos estáticos

// TODO: Aquí irá nuestro middleware de manejo de errores

// Iniciamos el servidor

app.listen(PORT, () => {

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

});
```

3. **Guarden el archivo.**
4. **Prueben que el servidor inicia:** En la terminal, ejecuten `node app.js`. Deberían ver el mensaje "Servidor corriendo en http://localhost:3000". Déjenlo corriendo o ciérrenlo por ahora (`Ctrl + C`).

Paso 2: Configuración de la Base de Datos y Modelo

Usaremos Sequelize para definir cómo se ven nuestros "Pilotos" y para interactuar con una base de datos SQLite simple.

1. **Creen una carpeta** llamada `db` en la raíz del proyecto.

2. **Dentro de `db`**, creen un archivo `db.js`.

3. **Peguen el siguiente código en `db.js`:**

```
// db/db.js

const { Sequelize, DataTypes } = require('sequelize');

// Configuración de Sequelize para usar SQLite

const sequelize = new Sequelize({

  dialect: 'sqlite',

  storage: 'database.sqlite' // El archivo de la base de datos se creará aquí

});
```

```
// Definimos el modelo Piloto
const Piloto = sequelize.define('Piloto', {
  nombre: {
    type: DataTypes.STRING,
    allowNull: false
  },
  escuderia: {
    type: DataTypes.STRING,
    allowNull: false
  },
  pais: {
    type: DataTypes.STRING,
    allowNull: true // Permitimos que el país sea opcional
  },
  numeroCoche: {
    type: DataTypes.INTEGER,
    allowNull: true, // Permitimos que el número sea opcional
    unique: true // Aseguramos que no haya dos pilotos con el mismo número (en teoría)
  },
  campeonatos: {
    type: DataTypes.INTEGER,
    defaultValue: 0 // Por defecto, 0 campeonatos
```

```
  }
});

// Función para conectar y sincronizar la base de datos
const connectDB = async () => {
  try {
    await sequelize.authenticate();
    console.log('Conexión a la base de datos SQLite establecida correctamente.');
    // Sincroniza los modelos con la base de datos (crea la tabla si no existe)
    await sequelize.sync({ alter: true }); // 'alter: true' intenta adaptar la tabla si el modelo cambia
    console.log('Modelos sincronizados con la base de datos.');
  } catch (error) {
    console.error('Error al conectar o sincronizar la base de datos:', error);
    // Aquí podrías decidir si quieres que la app termine si la DB falla
    process.exit(1); // Termina la aplicación con código de error
  }
};

module.exports = { sequelize, Piloto, connectDB };
```

4. **Guarden el archivo.**

5. **Integren la conexión en `app.js`:**

    - Al principio de `app.js`, importen la función `connectDB`.
    - Antes de `app.listen`, llamen a `connectDB`.

```
// app.js
```

```
const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000;

const { connectDB } = require('./db/db'); // <--- Importamos la función

// Middleware para parsear JSON en el cuerpo de las peticiones POST/PUT

app.use(express.json());

// Middleware para parsear datos de formularios (si usáramos form-urlencoded)

app.use(express.urlencoded({ extended: true }));

// TODO: Aquí irán nuestros middleware personalizados

// TODO: Aquí irán nuestras rutas API

// TODO: Aquí serviremos archivos estáticos

// TODO: Aquí irá nuestro middleware de manejo de errores

// Conectar a la base de datos ANTES de iniciar el servidor

connectDB().then(() => { // <--- Conectamos y luego iniciamos el servidor

  // Iniciamos el servidor

  app.listen(PORT, () => {

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

  });

}).catch(err => {

  console.error("Falló la conexión a la base de datos. Saliendo...", err);

  process.exit(1); // Salir si la DB no conecta

});
```

6. **Guarden `app.js`.**

7. **Prueben la conexión:** Ejecuten `node app.js`. Deberían ver los mensajes de conexión y sincronización exitosa, y luego el mensaje del servidor. Noten que se habrá creado un archivo `database.sqlite` en la raíz del proyecto. Ciérrenlo (`Ctrl + C`).

Paso 3: Implementación de Rutas API con Router

Vamos a crear un módulo de rutas específico para los pilotos usando `express.Router`.

1. **Creen una carpeta** llamada `routes` en la raíz del proyecto.

2. **Dentro de `routes`**, creen un archivo `pilotos.js`.

3. **Peguen el siguiente código en `pilotos.js`:**

```
// routes/pilotos.js

const express = require('express');

const router = express.Router();

const { Piloto } = require('../db/db'); // Importamos el modelo Piloto

// Ruta para obtener todos los pilotos

// GET /api/pilotos

router.get('/', async (req, res) => {

  try {

    const pilotos = await Piloto.findAll(); // Busca todos los pilotos en la DB

    res.json(pilotos); // Devuelve la lista como JSON

  } catch (error) {

    console.error("Error al obtener pilotos:", error);

    // Importante: Propagamos el error al middleware de manejo de errores
```

```
    res.status(500).json({ error: 'Error al obtener pilotos' });

  }

});

// Ruta para crear un nuevo piloto

// POST /api/pilotos

router.post('/', async (req, res) => {

  try {

    const nuevoPiloto = await Piloto.create(req.body); // Crea un nuevo piloto con los datos del body

    res.status(201).json(nuevoPiloto); // Devuelve el piloto creado con status 201 (Created)

  } catch (error) {

    console.error("Error al crear piloto:", error);

     // Importante: Propagamos el error al middleware de manejo de errores

    res.status(400).json({ error: error.message }); // Devuelve un error 400 si hay problemas con los datos

  }

});

// TODO: Aquí podríamos agregar rutas para GET individual, PUT, DELETE

module.exports = router; // Exportamos el router
```

4. **Guarden `pilotos.js`.**

5. **Integren el router en `app.js`:**

    - Importen el router.
    - Úsenlo con `app.use`.

```
// app.js

const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000;

const { connectDB } = require('./db/db');

const pilotosRouter = require('./routes/pilotos'); // <--- Importamos el router

// Middleware para parsear JSON en el cuerpo de las peticiones POST/PUT

app.use(express.json());

// Middleware para parsear datos de formularios (si usáramos form-urlencoded)

app.use(express.urlencoded({ extended: true }));

// TODO: Aquí irán nuestros middleware personalizados

// Usamos el router de pilotos para las rutas que empiecen con /api/pilotos

app.use('/api/pilotos', pilotosRouter); // <--- Usamos el router

// TODO: Aquí serviremos archivos estáticos

// TODO: Aquí irá nuestro middleware de manejo de errores

// Conectar a la base de datos ANTES de iniciar el servidor

connectDB().then(() => {

  // Iniciamos el servidor

  app.listen(PORT, () => {

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

  });

}).catch(err => {
```

```
  console.error("Falló la conexión a la base de datos. Saliendo...", err);
  process.exit(1);
});
```

6. **Guarden `app.js`.**
7. **Prueben las rutas (opcional, con Postman/Insomnia o `curl`):**
    - Inicien el servidor (`node app.js`).
    - `GET http://localhost:3000/api/pilotos` (debería devolver un array vacío `[]`)
    - `POST http://localhost:3000/api/pilotos` con body raw JSON:

      ```
      {
        "nombre": "Max Verstappen",
        "escuderia": "Red Bull Racing",
        "pais": "Países Bajos",
        "numeroCoche": 1,
        "campeonatos": 3
      }
      ```

      (debería devolver el objeto piloto creado con un ID).
    - `GET http://localhost:3000/api/pilotos` de nuevo (debería devolver el piloto creado).

Paso 4: Implementación de Middleware

Ahora vamos a añadir nuestros middleware personalizados y el de CORS. ¡Presten atención al orden!

1. **Creen una carpeta** llamada `middleware` en la raíz del proyecto.

2. **Dentro de `middleware`**, creen los siguientes archivos:

    - `logger.js`
    - `cleaner.js`
    - `errorHandler.js`

3. **Implementen `logger.js`:** Este middleware simplemente registrará cada petición que llegue.

```
// middleware/logger.js

const logger = (req, res, next) => {

  const start = Date.now();

  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

  // Aquí podríamos añadir lógica para cuando la respuesta termina

  res.on('finish', () => {

    const duration = Date.now() - start;

    console.log(`[${new Date().toISOString()}] ${req.method} ${req.url} - Status: ${res.statusCode} - Time: ${duration}ms`);

  });

  next(); // ¡Importante! Llama al siguiente middleware o ruta

};

module.exports = logger;
```

4. **Implementen `cleaner.js`:** Este middleware realizará una limpieza básica, como eliminar espacios en blanco al principio y al final de las cadenas en el `req.body`.

```
// middleware/cleaner.js

const cleaner = (req, res, next) => {

  if (req.body && typeof req.body === 'object') {
```

```
    for (const key in req.body) {
      if (typeof req.body[key] === 'string') {
        // Eliminar espacios en blanco al inicio y final
        req.body[key] = req.body[key].trim();
        // Opcional: Podríamos añadir más limpieza o validación aquí
      }
    }
  }
  next(); // ¡Importante! Llama al siguiente middleware o ruta
};
module.exports = cleaner;
```

5. **Implementen `errorHandler.js`:** Este middleware *siempre* debe ir al final de la cadena de middleware/rutas. Recibe un argumento `err` extra.

```
// middleware/errorHandler.js
const errorHandler = (err, req, res, next) => {
  console.error("----- ERROR CAPTURADO POR MIDDLEWARE -----");
  console.error(err.stack); // Registra el stack trace del error en consola
  // Determina el status y mensaje del error
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Ocurrió un error interno en el servidor';
  // Envía la respuesta de error al cliente
  res.status(statusCode).json({
    error: message,
```

```
    // Opcional: No enviar detalles sensibles del error en producción
    // stack: process.env.NODE_ENV === 'development' ? err.stack : undefined
  });
};

module.exports = errorHandler;
```

6. **Integren los middleware en `app.js`:** El **orden** es crucial.
    - El logger va primero para registrar todas las peticiones.
    - CORS va antes de las rutas para permitir peticiones desde el frontend (aunque en este caso servimos HTML directamente, es buena práctica incluirlo si pensamos en un frontend separado).
    - `express.json()` y `urlencoded()` deben ir *antes* del cleaner porque el cleaner opera sobre `req.body`, que es populado por estos middleware.
    - El cleaner va antes de las rutas para limpiar los datos antes de que lleguen a la lógica de negocio.
    - El router va después de los middleware generales.
    - El `errorHandler` va *al final de todo*, después de todas las rutas y otros `app.use`.

```
// app.js

const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000;

const cors = require('cors'); // <--- Importamos CORS

const { connectDB } = require('./db/db');

const pilotosRouter = require('./routes/pilotos');

// --- Importamos nuestros middleware ---

const loggerMiddleware = require('./middleware/logger');
```

```
const cleanerMiddleware = require('./middleware/cleaner');

const errorHandlerMiddleware = require('./middleware/errorHandler');

// ----------------------------------

// --- Middleware Generales (Orden Importa) ---

app.use(loggerMiddleware); // 1. Primero el logger

app.use(cors());           // 2. Luego CORS (permite peticiones cruzadas)

app.use(express.json());   // 3. Parsing de JSON (necesario antes del cleaner y rutas POST/PUT)

app.use(express.urlencoded({ extended: true })); // 4. Parsing de URL-encoded (si aplica)

app.use(cleanerMiddleware); // 5. Limpieza de datos de entrada

// ----------------------------------------

// TODO: Aquí serviremos archivos estáticos (Lo haremos en el siguiente paso)

// --- Rutas API ---

app.use('/api/pilotos', pilotosRouter); // Nuestras rutas específicas

// -----------------

// --- Middleware de Manejo de Errores (¡SIEMPRE AL FINAL!) ---

app.use(errorHandlerMiddleware);

// ------------------------------------------------------------


// Conectar a la base de datos ANTES de iniciar el servidor

connectDB().then(() => {

  // Iniciamos el servidor

  app.listen(PORT, () => {
```

```
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
  });
}).catch(err => {
  console.error("Falló la conexión a la base de datos. Saliendo...", err);
  process.exit(1);
});
```

7. **Guarden `app.js` y los archivos de middleware.**

8. **Prueben los middleware:**

    - Inicien el servidor (`node app.js`).
    - Realicen algunas peticiones (GET y POST) a `/api/pilotos` usando Postman/Insomnia o `curl`.
    - Observen la terminal donde corre `node app.js`. Deberían ver los logs de cada petición.
    - Prueben a enviar un piloto con espacios extra en el nombre o escudería (`" Checo Pérez  "`). Al recuperar la lista, ¿se limpiaron? (Sí, deberían haberse limpiado al crear/guardar).
    - Para probar el error handler, pueden modificar temporalmente la ruta POST en `routes/pilotos.js` para que lance un error intencionalmente (ej: `throw new Error("Simulando un error al guardar");`). Realicen un POST y vean si el middleware de error lo captura y responde con status 500 y el mensaje. Luego, reviertan el cambio.

Paso 5: Frontend Sencillo con HTML, Bootstrap y Vanilla JS

Ahora vamos a crear una página HTML simple que use Bootstrap para el estilo y Vanilla JS para interactuar con nuestro backend.

1. **Creen una carpeta** llamada `public` en la raíz del proyecto. Aquí irán nuestros archivos estáticos (HTML, CSS, JS del frontend).

2. **Dentro de `public`**, creen un archivo `index.html`.

3. **Peguen el siguiente código en `index.html`:** Usamos Bootstrap vía CDN para no tener que instalarlo localmente.

```
<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Boxes & Pitstops - Pilotos F1</title>

    <!-- Bootstrap CSS CDN -->

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">

    <style>

        body {

            padding-top: 20px;

        }

    </style>

</head>

<body>

    <div class="container">

        <h1 class="mb-4 text-center">🏆 Boxes & Pitstops 🏁</h1>

        <p class="text-center">Gestión básica de Pilotos de F1</p>

        <hr>

        <h2>Lista de Pilotos</h2>
```

```
<ul id="pilotos-list" class="list-group mb-4">
    <!-- Los pilotos se cargarán aquí con JavaScript -->
    <li class="list-group-item text-center text-muted">Cargando pilotos...</li>
</ul>
<hr>
<h2>Agregar Nuevo Piloto</h2>
<form id="add-piloto-form">
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" class="form-control" id="nombre" required>
    </div>
    <div class="mb-3">
        <label for="escuderia" class="form-label">Escudería</label>
        <input type="text" class="form-control" id="escuderia" required>
    </div>
    <div class="mb-3">
        <label for="pais" class="form-label">País</label>
        <input type="text" class="form-control" id="pais">
    </div>
    <div class="mb-3">
        <label for="numeroCoche" class="form-label">Número de Coche</label>
        <input type="number" class="form-control" id="numeroCoche">
```

```
            </div>

             <div class="mb-3">

                <label for="campeonatos" class="form-label">Campeonatos
Mundiales</label>

                <input type="number" class="form-control" id="campeonatos" value="0">

            </div>

            <button type="submit" class="btn btn-primary">Agregar Piloto</button>

        </form>

    </div>

    <!-- Bootstrap Bundle with Popper -->

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlp
cV8Qyq46cDfL" crossorigin="anonymous"></script>

    <!-- Nuestro script JS -->

    <script src="/script.js"></script>

</body>

</html>
```

4. **Dentro de `public`**, creen un archivo `script.js`.

5. **Peguen el siguiente código en `script.js`:**

```
// public/script.js

document.addEventListener('DOMContentLoaded', () => {

    const pilotosList = document.getElementById('pilotos-list');

    const addPilotoForm = document.getElementById('add-piloto-form');
```

```
    // Función para cargar y mostrar los pilotos
    const fetchPilotos = async () => {
        try {
            const response = await fetch('/api/pilotos'); // Llama a nuestra API
            if (!response.ok) {
                throw new Error(`Error HTTP: ${response.status}`);
            }
            const pilotos = await response.json(); // Parsear la respuesta JSON
            // Limpiar la lista actual
            pilotosList.innerHTML = '';
            if (pilotos.length === 0) {
                pilotosList.innerHTML = '<li class="list-group-item text-center text-muted">No hay pilotos cargados todavía.</li>';
            } else {
                // Mostrar cada piloto en la lista
                pilotos.forEach(piloto => {
                    const li = document.createElement('li');
                    li.classList.add('list-group-item');
                    li.innerHTML = `
                        <strong>${piloto.nombre}</strong> (${piloto.escuderia}) - ${piloto.pais || 'N/A'} | #${piloto.numeroCoche || 'N/A'} | ${piloto.campeonatos} Campeonatos
                        <!-- Opcional: Agregar botones de editar/eliminar aquí -->
                    `;
```

```
                pilotosList.appendChild(li);

            });

        }

    } catch (error) {

        console.error('Error al cargar los pilotos:', error);

        pilotosList.innerHTML = `<li class="list-group-item text-danger">Error al cargar pilotos: ${error.message}</li>`;

    }

};

// Manejar el envío del formulario

addPilotoForm.addEventListener('submit', async (event) => {

    event.preventDefault(); // Evitar que la página se recargue

    // Obtener datos del formulario

    const nombre = document.getElementById('nombre').value;

    const escuderia = document.getElementById('escuderia').value;

    const pais = document.getElementById('pais').value;

    const numeroCoche = document.getElementById('numeroCoche').value;

     const campeonatos = document.getElementById('campeonatos').value;

    const nuevoPiloto = {

        nombre,

        escuderia,

        pais: pais || null, // Enviar null si está vacío
```

```
            numeroCoche: numeroCoche ? parseInt(numeroCoche, 10) : null, // Convertir a
número o null

            campeonatos: campeonatos ? parseInt(campeonatos, 10) : 0

        };

        try {

            const response = await fetch('/api/pilotos', {

                method: 'POST',

                headers: {

                    'Content-Type': 'application/json',

                },

                body: JSON.stringify(nuevoPiloto), // Enviar los datos como JSON

            });

            const result = await response.json();

            if (!response.ok) {

                // Si la respuesta no es OK, lanzar un error con el mensaje del backend

                const errorMessage = result.error || 'Error al agregar piloto';

                throw new Error(`Error al agregar piloto: ${response.status} -
${errorMessage}`);

            }

            console.log('Piloto agregado:', result);

            // Limpiar el formulario

            addPilotoForm.reset();

            // Recargar la lista de pilotos para mostrar el nuevo
```

```
            fetchPilotos();
        } catch (error) {
            console.error('Error al agregar el piloto:', error);
            // Mostrar un mensaje de error al usuario (simple alerta por ahora)
            alert(`No se pudo agregar el piloto: ${error.message}`);
        }
    });
    // Cargar los pilotos cuando la página se cargue
    fetchPilotos();
});
```

6. **Guarden `index.html` y `script.js`.**

7. **Sirvan los archivos estáticos desde `app.js`:** Añadan el middleware `express.static` *antes* de las rutas API, pero *después* de los middleware generales como logger, cors, json, urlencoded y cleaner (ya que estos últimos podrían ser necesarios para ciertas peticiones, aunque `static` las suele interceptar antes si encuentra el archivo).

```
// app.js
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;
const cors = require('cors');
const { connectDB } = require('./db/db');
const pilotosRouter = require('./routes/pilotos');
const loggerMiddleware = require('./middleware/logger');
```

```
const cleanerMiddleware = require('./middleware/cleaner');
const errorHandlerMiddleware = require('./middleware/errorHandler');
// --- Middleware Generales (Orden Importa) ---
app.use(loggerMiddleware); // 1. Primero el logger
app.use(cors());           // 2. Luego CORS (permite peticiones cruzadas)
app.use(express.json());   // 3. Parsing de JSON
app.use(express.urlencoded({ extended: true })); // 4. Parsing de URL-encoded
app.use(cleanerMiddleware); // 5. Limpieza de datos de entrada
// ----------------------------------------
// --- Servir archivos estáticos ---
app.use(express.static('public')); // <--- Sirve los archivos de la carpeta 'public'
// --------------------------------
// --- Rutas API ---
app.use('/api/pilotos', pilotosRouter); // Nuestras rutas específicas
// -----------------
// --- Middleware de Manejo de Errores (¡SIEMPRE AL FINAL!) ---
app.use(errorHandlerMiddleware);
// ------------------------------------------------------------

// Conectar a la base de datos ANTES de iniciar el servidor
connectDB().then(() => {
  // Iniciamos el servidor
  app.listen(PORT, () => {
```

```
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
    console.log(`Frontend disponible en http://localhost:${PORT}`); // Mensaje útil
  });
}).catch(err => {
  console.error("Falló la conexión a la base de datos. Saliendo...", err);
  process.exit(1);
});
```

8. **Guarden `app.js`.**
9. **Prueben la aplicación completa:**
    - Inicien el servidor (`node app.js`).
    - Abran su navegador web y vayan a `http://localhost:3000`.
    - Deberían ver la página de Bootstrap. La lista de pilotos inicialmente estará vacía (o mostrará los que agregaron antes con Postman).
    - Usen el formulario para agregar nuevos pilotos (ej: "Checo Pérez", "Sergio Pérez ", "Fernando Alonso", etc.).
    - Observen la lista cómo se actualiza.
    - Observen la terminal de VS Code para ver los logs de las peticiones GET y POST.
    - Intenten agregar un piloto con un número de coche que ya exista (si agregaron uno con Postman, por ejemplo). Deberían ver un error en la consola del navegador y quizás una alerta (dependiendo de cómo implementaron el manejo de error en el frontend, si no, sólo en la consola del backend). Esto muestra cómo los errores del backend se pueden propagar y manejar.

Cierre

En esta ejercitación, hemos montado un servidor Express completo, integrado una base de datos simple con Sequelize, creado rutas API organizadas con `express.Router`, y lo más importante: hemos añadido y comprendido la función de varios middleware clave:

- **Logger:** Para ver qué está pasando en nuestro servidor.
- **CORS:** Para permitir que clientes de otros orígenes accedan a nuestra API (aunque aquí el frontend es servido por el mismo server).

- **Cleaner:** Para sanitizar datos de entrada básicos.
- **Error Handler:** Para tener un punto central donde gestionar y responder a los errores inesperados.

También hemos visto cómo un frontend simple de HTML/Bootstrap/Vanilla JS puede consumir nuestra API REST.

Recuerden que los middleware son súper poderosos y son el corazón de muchas funcionalidades en Express. El orden en que los definen es crítico.