**Actividad de Clase**

**Aplicación de Callbacks, Promesas, Async y Await**

El objetivo de esta actividad es desarrollar ejercicios de aplicación usando el entorno de Node JS y el Framework Express.js, para aprender el uso de Callbacks, Promesas, Async y Await.

# Desarrollo

Para desarrollar la actividad lo primero que haremos es abrir nuestro **Visual Studio Code**, y abriremos una carpeta donde desarrollaremos nuestros ejercicios, para este caso he creado la carpeta llamada **Ejercicios**:

Vamos a crear una carpeta llamada **mi-app**, para ello usamos este comando:

mkdir mi-app

Luego con el siguiente comando ingresaremos dentro de la carpeta desde la consola:

cd mi-app

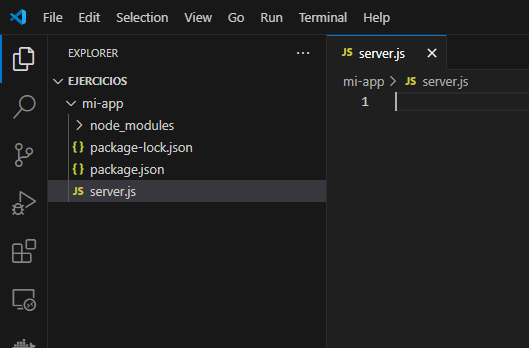
Luego Ejecutaremos este código para crear un proyecto node.js:

npm init -y

Ahora vamos a crear una aplicación de prueba usando el framewrok express js, para instalarlo ejecutamos el siguiente código en la consola:

npm install express --save

Seguidamente, vamos a crear el archivo **server.js**



Por último, editaremos el archivo **package.json**, para poder ejecutar nuestra app sin problemas:

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "node server.js",

    "dev": "nodemon server.js"

  }

Para ejecutar la aplicación, en nuestra consola de Visual Studio Code, colocamos el siguiente código:

npm start  # Ejecuta con Node.js

npm run dev  # Ejecuta con Nodemon (recarga automática)

|  |  |  |  |
| --- | --- | --- | --- |
| **Comando** | **Ejecuta con** | **Recarga automática** | **Definición en package.json** |
| npm start | Node.js | ❌ No | "start": "node server.js" |
| npm run dev | Nodemon | ✅ Sí | "dev": "nodemon server.js" |

La segunda forma de ejecutar utiliza el comando **Nodemon**, para que esto funcione correctamente debemos instalarlo con los siguientes comandos:

**Instalación global (disponible en todo el sistema)**

npm install -g nodemon

Esto permite usar nodemon en cualquier terminal:

**Instalación local en el proyecto (solo disponible dentro de la carpeta del proyecto)**

npm install --save-dev nodemon

**CALLBACKS**

Crear un servicio GET en Express.js usando exclusivamente Callbacks para consultar una lista de usuarios, para ello dentro de nuestro archivo **server.js**, colocamos este código:

const express = require("express"); // Importamos Express.js

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

const PORT = 3000; // Definimos el puerto del servidor

// Simulación de una base de datos de usuarios

const usuarios = [

  { id: 1, nombre: "Juan Pérez", email: "juan@example.com" },

  { id: 2, nombre: "María López", email: "maria@example.com" },

  { id: 3, nombre: "Carlos Sánchez", email: "carlos@example.com" }

];

// Función para obtener usuarios con Callback

function obtenerUsuarios(callback) {

  setTimeout(() => {

    callback(null, usuarios); // Llamamos al callback con los datos

  }, 2000); // Simula una consulta a base de datos con 2 segundos de demora

}

// Ruta GET para obtener los usuarios

app.get("/usuarios", (req, res) => {

  obtenerUsuarios((error, listaUsuarios) => {

    if (error) {

      return res.status(500).json({ error: "Error al obtener usuarios" });

    }

    res.json(listaUsuarios); // Retorna la lista de usuarios en formato JSON

  });

});

// Iniciar el servidor

app.listen(PORT, () => {

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

});

**Explicación del código:**

const express = require("express"); // Importamos Express.js

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

const PORT = 3000; // Definimos el puerto del servidor

* **require("express")** importa el framework Express.js.
* **app = express()** inicializa una aplicación Express.
* **PORT = 3000** define el puerto en el que correrá el servidor.

function obtenerUsuarios(callback) {

  setTimeout(() => {

    callback(null, usuarios); // Llamamos al callback con los datos

  }, 2000); // Simula una consulta a base de datos con 2 segundos de demora

}

* La función **obtenerUsuarios(callback)** recibe un **callback** como argumento.
* Usamos **setTimeout()** para simular una consulta a la base de datos con una demora de 2 segundos.
* Luego, llamamos al **callback** con **null** como error y enviamos usuarios como resultado.

// Ruta GET para obtener los usuarios

app.get("/usuarios", (req, res) => {

  obtenerUsuarios((error, listaUsuarios) => {

    if (error) {

      return res.status(500).json({ error: "Error al obtener usuarios" });

    }

    res.json(listaUsuarios); // Retorna la lista de usuarios en formato JSON

  });

});

* **app.get("/usuarios", callback)** define una ruta **GET** en Express.js.
* Llamamos a **obtenerUsuarios()** y pasamos un **callback** que maneja la respuesta.
* Si ocurre un error, respondemos con un estado **500** y un mensaje de error.
* Si todo va bien, enviamos la lista de usuarios en formato **JSON** con **res.json().**

**PROMESAS**

Replicaremos el mismo ejercicio usando promesas en lugar de callbacks, agregamos este código a nuestro archivo **server.js**:

const express = require("express"); // Importamos Express.js

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

const PORT = 3000; // Definimos el puerto del servidor

// Simulación de una base de datos de usuarios

const usuarios = [

  { id: 1, nombre: "Juan Pérez", email: "juan@example.com" },

  { id: 2, nombre: "María López", email: "maria@example.com" },

  { id: 3, nombre: "Carlos Sánchez", email: "carlos@example.com" }

];

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios); // Resolvemos la promesa con la lista de usuarios

      } else {

        reject("No hay usuarios disponibles"); // Rechazamos la promesa si no hay datos

      }

    }, 2000); // Simulamos una consulta a base de datos con 2 segundos de demora

  });

}

// Ruta GET para obtener los usuarios usando Promesas

app.get("/usuarios", (req, res) => {

  obtenerUsuarios()

    .then(listaUsuarios => res.json(listaUsuarios)) // Si la promesa se resuelve, enviamos los datos

    .catch(error => res.status(500).json({ error })); // Si hay un error, lo manejamos con .catch()

});

// Iniciar el servidor

app.listen(PORT, () => {

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

});

**Explicación del código:**

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios); // Resolvemos la promesa con la lista de usuarios

      } else {

        reject("No hay usuarios disponibles"); // Rechazamos la promesa si no hay datos

      }

    }, 2000); // Simulamos una consulta a base de datos con 2 segundos de demora

  });

}

* **new Promise((resolve, reject) => { ... })** crea una promesa.
* Si hay usuarios en la base de datos, llamamos a **resolve(usuarios)** para devolverlos.
* Si no hay usuarios, llamamos a **reject("No hay usuarios disponibles"),** que se manejará en **.catch().**
* Simulamos un retardo de 2 segundos con **setTimeout()** para representar una consulta a base de datos.

// Ruta GET para obtener los usuarios usando Promesas

app.get("/usuarios", (req, res) => {

  obtenerUsuarios()

    .then(listaUsuarios => res.json(listaUsuarios)) // Si la promesa se resuelve, enviamos los datos

    .catch(error => res.status(500).json({ error })); // Si hay un error, lo manejamos con .catch()

});

* Llamamos a la función **obtenerUsuarios(),** que devuelve una Promesa.
* Si la Promesa se resuelve con éxito **(resolve()),** el **.then()** enviará la lista de usuarios como respuesta **JSON**.
* Si ocurre un error **(reject()),** el **.catch()** manejará el error y enviará un estado HTTP 500 (Internal Server Error) con un mensaje JSON.

**Beneficios de Usar Promesas en Express.js**

* Código más limpio y legible, evitando la anidación excesiva de callbacks.
* Mejor manejo de errores con .catch(), evitando el callback hell.
* Encadenamiento de procesos más estructurado, permitiendo múltiples operaciones asíncronas sin complicaciones.

**Async y Await**

Por último, mejoraremos el mismo método usando Async y Await, para ello agregamos este código a nuestro archivo **server.js**:

const express = require("express"); // Importamos Express.js

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

const PORT = 3000; // Definimos el puerto del servidor

// Simulación de una base de datos de usuarios

const usuarios = [

  { id: 1, nombre: "Juan Pérez", email: "juan@example.com" },

  { id: 2, nombre: "María López", email: "maria@example.com" },

  { id: 3, nombre: "Carlos Sánchez", email: "carlos@example.com" }

];

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios);

      } else {

        reject(new Error("No hay usuarios disponibles"));

      }

    }, 2000);

  });

}

// Ruta GET optimizada usando Async/Await

app.get("/usuarios", async (req, res) => {

  try {

    const listaUsuarios = await obtenerUsuarios(); // Esperamos el resultado de la promesa

    res.json(listaUsuarios); // Enviamos la respuesta en formato JSON

  } catch (error) {

    res.status(500).json({ error: error.message }); // Manejamos el error con try/catch

  }

});

// Iniciar el servidor

app.listen(PORT, () => {

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

});

**Explicación del código:**

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios);

      } else {

        reject(new Error("No hay usuarios disponibles"));

      }

    }, 2000);

  });

}

* Función que devuelve una **Promesa**, simulando una consulta a base de datos.
* Usamos **reject(new Error("mensaje"))** en lugar de solo **reject("mensaje")** para manejar mejor los errores.

// Ruta GET optimizada usando Async/Await

app.get("/usuarios", async (req, res) => {

  try {

    const listaUsuarios = await obtenerUsuarios(); // Esperamos el resultado de la promesa

    res.json(listaUsuarios); // Enviamos la respuesta en formato JSON

  } catch (error) {

    res.status(500).json({ error: error.message }); // Manejamos el error con try/catch

  }

});

* Convertimos la función en asíncrona con **async (req, res) => {}.**
* Usamos **await** **obtenerUsuarios();** en lugar de **.then(),** lo que hace que el código sea más limpio.
* Manejamos los errores con **try/catch**, de forma similar a código síncrono.

**¿Por qué es mejor usar Async/Await?**

* **Código más limpio y fácil de leer:** Se ve como código síncrono y evita .then() y .catch().
* **Manejo de errores más robusto:** Con try/catch, los errores se manejan de forma natural.
* **Mejor organización del flujo de ejecución:** await detiene la ejecución hasta que la Promesa se resuelve, evitando encadenamientos confusos.
* **Depuración más sencilla:** Los errores se capturan correctamente y se muestran con un stack trace más claro.

**Conclusión:**

* **Async/Await** es la mejor opción para manejar asincronía en Express.js, ya que combina la eficiencia de las Promesas con la simplicidad del código síncrono.

**Desestructuración de Objetos**

Por último, modificaremos el método de tal forma que podamos aplicar desestructuración de objetos, para ello agregamos este código a nuestro archivo **server.js**:

const express = require("express"); // Importamos Express.js

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

const PORT = 3000; // Definimos el puerto del servidor

// Simulación de una base de datos de usuarios

const usuarios = [

  { id: 1, nombre: "Juan Pérez", email: "juan@example.com" },

  { id: 2, nombre: "María López", email: "maria@example.com" },

  { id: 3, nombre: "Carlos Sánchez", email: "carlos@example.com" }

];

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios);

      } else {

        reject(new Error("No hay usuarios disponibles"));

      }

    }, 2000);

  });

}

// Ruta GET con Desestructuración

app.get("/usuarios", async (req, res) => {

  try {

    const listaUsuarios = await obtenerUsuarios(); // Obtenemos la lista de usuarios

    // Usamos desestructuración para extraer cada usuario y solo devolver ciertos campos

    const usuariosFormateados = listaUsuarios.map(({ id, nombre }) => ({

      id,

      nombre

    }));

    res.json(usuariosFormateados);

  } catch ({ message }) { // Desestructuramos el error para obtener solo el mensaje

    res.status(500).json({ error: message });

  }

});

// Iniciar el servidor

app.listen(PORT, () => {

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

});

**Explicación del código:**

// Función para obtener usuarios con Promesas

function obtenerUsuarios() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      if (usuarios.length > 0) {

        resolve(usuarios);

      } else {

        reject(new Error("No hay usuarios disponibles"));

      }

    }, 2000);

  });

}

* Función que devuelve una **Promesa**, simulando una consulta a base de datos.
* Si hay usuarios, resolvemos la promesa con **resolve(usuarios).**
* Si no hay usuarios, lanzamos un error con **reject(new Error("mensaje")).**

// Ruta GET con Desestructuración

app.get("/usuarios", async (req, res) => {

  try {

    const listaUsuarios = await obtenerUsuarios(); // Obtenemos la lista de usuarios

    // Usamos desestructuración para extraer cada usuario y solo devolver ciertos campos

    const usuariosFormateados = listaUsuarios.map(({ id, nombre }) => ({

      id,

      nombre

    }));

    res.json(usuariosFormateados);

  } catch ({ message }) { // Desestructuramos el error para obtener solo el mensaje

    res.status(500).json({ error: message });

  }

});

* **const listaUsuarios = await obtenerUsuarios();** → Usamos await para esperar la Promesa.
* **const usuariosFormateados = listaUsuarios.map(({ id, nombre }) => ({ id, nombre }));**
  + Usamos desestructuración en **map()** para extraer solo **id** y **nombre** de cada usuario.
  + Evitamos devolver datos sensibles como el **email**.
    - **catch ({ message }) →** Desestructuramos el objeto error para obtener solo message.

**Registro de Productos con Promesas y Async/Await**

Desarrollar una API REST con Express.js que permita gestionar un registro de productos. Los productos deben tener los siguientes campos:

* id (número autoincremental)
* nombre (string)
* descripcion (string)
* stock (número)
* precio (número)

La API debe permitir:

* Registrar un nuevo producto (POST /productos)
* Obtener la lista de productos (GET /productos)
* Obtener un producto por ID (GET /productos/:id)
* Actualizar un producto (PUT /productos/:id)
* Eliminar un producto (DELETE /productos/:id)

Crear el archivo server.js y agregar el siguiente código base:

const express = require('express');

const app = express();

app.use(express.json()); // Middleware para leer JSON

// Importar módulo de base de datos

const productosDB = require('./productosDB');

// Ruta para obtener todos los productos

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

    try {

        const productos = await productosDB.obtenerProductos();

        res.json(productos);

    } catch (error) {

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

    }

});

// Ruta para obtener un producto por ID

app.get('/productos/:id', async (req, res) => {

    try {

        const id = parseInt(req.params.id);

        const producto = await productosDB.obtenerProductoPorId(id);

        if (!producto) {

            return res.status(404).json({ error: 'Producto no encontrado' });

        }

        res.json(producto);

    } catch (error) {

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

    }

});

// Ruta para agregar un nuevo producto

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

    try {

        const { nombre, descripcion, stock, precio } = req.body;

        if (!nombre || !descripcion || stock == null || precio == null) {

            return res.status(400).json({ error: 'Todos los campos son obligatorios' });

        }

        const nuevoProducto = await productosDB.agregarProducto({ nombre, descripcion, stock, precio });

        res.status(201).json(nuevoProducto);

    } catch (error) {

        res.status(500).json({ error: 'Error al agregar el producto' });

    }

});

// Ruta para actualizar un producto

app.put('/productos/:id', async (req, res) => {

    try {

        const id = parseInt(req.params.id);

        const { nombre, descripcion, stock, precio } = req.body;

        const productoActualizado = await productosDB.actualizarProducto(id, { nombre, descripcion, stock, precio });

        if (!productoActualizado) {

            return res.status(404).json({ error: 'Producto no encontrado' });

        }

        res.json(productoActualizado);

    } catch (error) {

        res.status(500).json({ error: 'Error al actualizar el producto' });

    }

});

// Ruta para eliminar un producto

app.delete('/productos/:id', async (req, res) => {

    try {

        const id = parseInt(req.params.id);

        const eliminado = await productosDB.eliminarProducto(id);

        if (!eliminado) {

            return res.status(404).json({ error: 'Producto no encontrado' });

        }

        res.json({ mensaje: 'Producto eliminado' });

    } catch (error) {

        res.status(500).json({ error: 'Error al eliminar el producto' });

    }

});

// Iniciar servidor

const PORT = 3000;

app.listen(PORT, () => {

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

});

Crear el archivo productosDB.js para simular una base de datos con Promesas:

let productos = [];

let idCounter = 1;

// Obtener todos los productos

function obtenerProductos() {

    return new Promise((resolve) => {

        setTimeout(() => resolve(productos), 500);

    });

}

// Obtener un producto por ID

function obtenerProductoPorId(id) {

    return new Promise((resolve) => {

        setTimeout(() => {

            const producto = productos.find(p => p.id === id);

            resolve(producto);

        }, 500);

    });

}

// Agregar un nuevo producto

function agregarProducto({ nombre, descripcion, stock, precio }) {

    return new Promise((resolve) => {

        setTimeout(() => {

            const nuevoProducto = { id: idCounter++, nombre, descripcion, stock, precio };

            productos.push(nuevoProducto);

            resolve(nuevoProducto);

        }, 500);

    });

}

// Actualizar un producto

function actualizarProducto(id, datos) {

    return new Promise((resolve) => {

        setTimeout(() => {

            const index = productos.findIndex(p => p.id === id);

            if (index === -1) return resolve(null);

            productos[index] = { ...productos[index], ...datos };

            resolve(productos[index]);

        }, 500);

    });

}

// Eliminar un producto

function eliminarProducto(id) {

    return new Promise((resolve) => {

        setTimeout(() => {

            const index = productos.findIndex(p => p.id === id);

            if (index === -1) return resolve(false);

            productos.splice(index, 1);

            resolve(true);

        }, 500);

    });

}

module.exports = { obtenerProductos, obtenerProductoPorId, agregarProducto, actualizarProducto, eliminarProducto };