

Manual Técnico

Creación del Proyecto

Equipo:

Sara Montserrat Oliva Roque

Nely Fernanda Sánchez Rivera

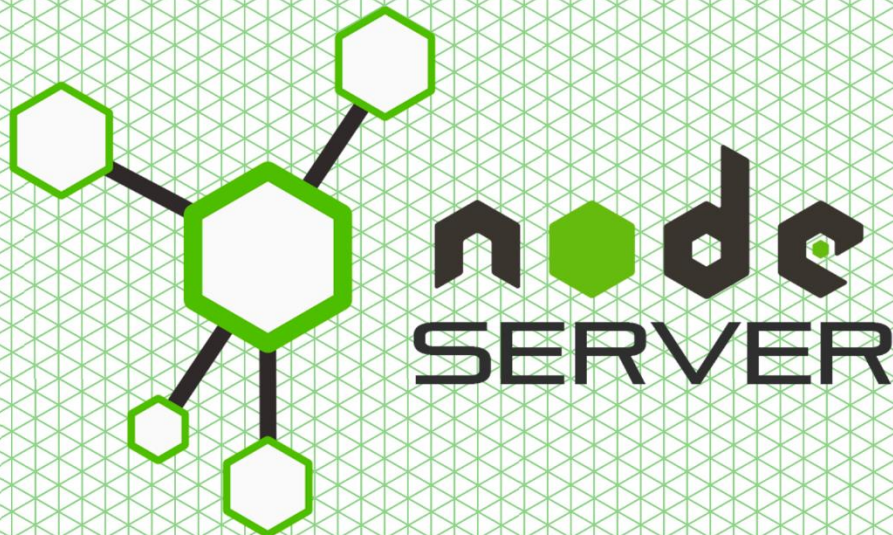
Erik Refugio Contreras López

Dino E Ramirez Hernandez

Universidad Autónoma de Aguascalientes

LITC 7

2023



Contenido

Requisitos.....	3
Configuración del Entorno:.....	3
Instalación de Dependencias:	3
Estructura del Proyecto:.....	4
Configuración de Variables de Entorno:.....	6
Configuraciones iniciales en package.json.....	7
Creación de main.js (arranque del servidor)	7
Configuración de Express y Middleware:.....	8
Creación de programs.js (Modelo de Mongoose):.....	10
Configuración de Rutas:.....	12
Creación y configuración de las vistas EJS (Embedded JavaScript):	14
Pruebas y Depuración:	15
Agradecimientos:.....	16

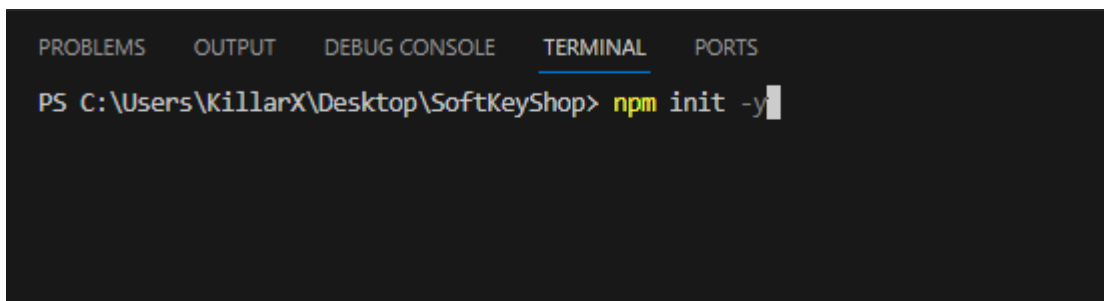
Requisitos

Node.js <https://nodejs.org/en/download>

Visual Studio Code <https://code.visualstudio.com/download>

Configuración del Entorno

- Se creó un nuevo proyecto **Node.js** la carpeta llamada **SoftKeyShop**.
- Se inicializó el proyecto con **npm init**.

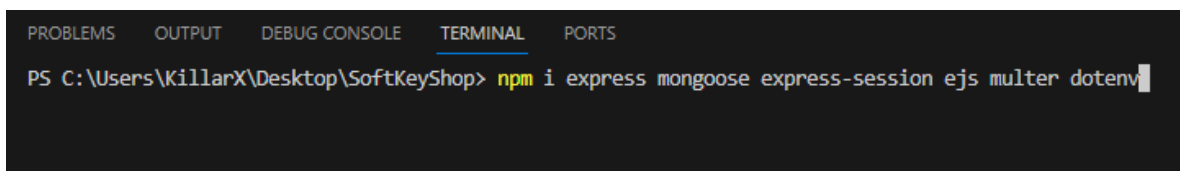


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\KillarX\Desktop\SoftKeyShop> npm init -y
```

Comando: npm init -y

Instalación de Dependencias

- Se instalaron varias dependencias utilizando **npm install**:
 - **express**: Para el desarrollo del servidor web.
 - **mongoose**: Para la conexión y manipulación de la base de datos MongoDB.
 - **ejs**: Un motor de plantillas para generar vistas HTML.
 - **express-session**: Para gestionar sesiones de usuario.
 - **multer**: Para el manejo de archivos y la carga de imágenes.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\KillarX\Desktop\SoftKeyShop> npm i express mongoose express-session ejs multer dotenv
```

Comando: npm i express mongoose express-session ejs multer dotenv

Estructura del Proyecto

- Se crearon directorios para organizar el código:
 - **models**: Contiene el modelo de datos para los programas (programs.js).
 - **uploads**: Almacena las imágenes cargadas por los usuarios.
 - **views**: Contiene las plantillas EJS para las diferentes páginas.
 - **routes**: Contiene las rutas y la lógica de manejo de solicitudes HTTP.

Directorio:

-models

programs.js

-routes

routes.js

-uploads

-views

about.ejs

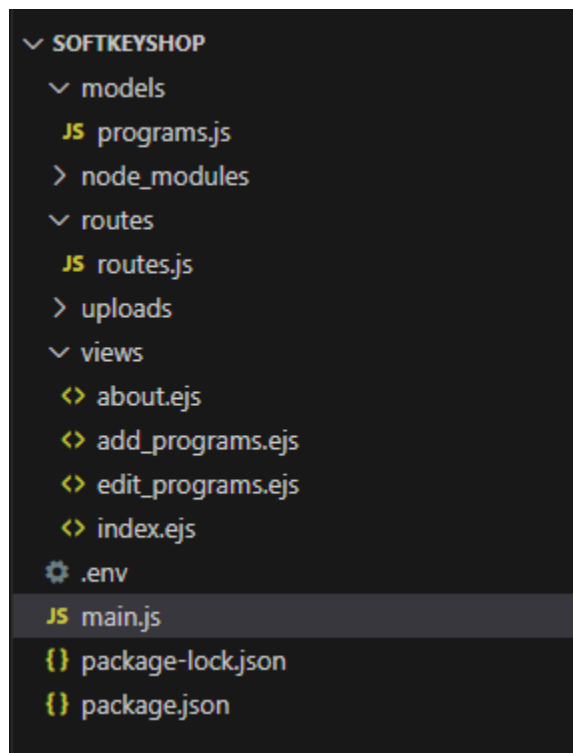
add_programs.ejs

edit_programs.ejs

index.ejs

.env

main.js



Explicación del directorio

Models/programs.js: En este archivo, defino el esquema del modelo de datos (Program) que uso para interactuar con mi base de datos MongoDB. Es esencial para asegurar que mis datos se almacenen y recuperen correctamente.

Uploads: Utilizo este directorio para almacenar las imágenes que los usuarios cargan al agregar o editar programas. Cada vez que alguien sube una imagen, esta se guarda en el directorio "uploads".

Views: En este directorio, guardo mis archivos de vistas en formato EJS. Cada archivo EJS representa una página específica de mi aplicación.

Views/about.ejs: Aquí presento información sobre el equipo.

Views/add_programs.ejs: Contiene un formulario para agregar nuevos programas con detalles como nombre, descripción, precio e imagen.

Views/edit_programs.ejs: Similar a add_programs.ejs, pero prellenado con los datos del programa seleccionado.

Views/index.ejs: La página principal que muestra una tabla con la lista de programas existentes.

.env: Este archivo almacena variables de entorno cruciales, como la URI de la base de datos. Es una forma segura de manejar configuraciones sensibles sin exponerlas directamente en el código.

Main.js: Este archivo contiene la configuración del servidor Express, la conexión a la base de datos, middleware, rutas y configuración del motor de plantillas EJS. Es el punto de entrada principal para mi aplicación.

Routes/routes.js: Este archivo define las rutas específicas de mi aplicación usando Express Router. Contiene lógica de manejo de solicitudes para operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en los programas. Aquí está la función de cada ruta en este archivo:

- POST '/agregar': Agrega un nuevo programa a la base de datos.
- GET '/': Recupera todos los programas de la base de datos y renderiza la página principal.
- GET '/edit/:id': Recupera un programa específico por ID y renderiza la página de edición con sus datos.
- POST '/actualizar/:id': Actualiza un programa existente en la base de datos.
- GET '/delete/:id': Elimina un programa de la base de datos.
- GET '/agregar': Renderiza la página para agregar nuevos programas.
- GET '/info': Renderiza la página de información sobre el equipo o la aplicación.

El archivo **routes.js** es crucial para la gestión de rutas y la lógica de la aplicación, separando estas responsabilidades para mantener un código más organizado y modular.

Configuración de Variables de Entorno

Se utilizó el paquete **dotenv** para cargar variables de entorno desde un archivo **.env**.

Variables de Entorno:

```
DB_URI = mongodb+srv://[user]:[pass]@cluster0.xxxx.mongodb.net/xxxx
PORT = 5000
```

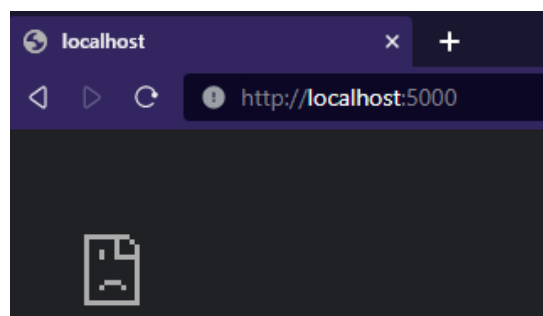
Se creo una cuenta de mongoDB, posteriormente se obtuvo la URI que se guardó en la variable de entorno DB_URI en .env

Obtener la URI y creación de base de datos:

1. Creé una cuenta en MongoDB:
 - Accedí al sitio web de MongoDB y creé una cuenta para comenzar a utilizar sus servicios.
2. Inicié sesión en MongoDB Atlas:
 - Después de crear mi cuenta, inicié sesión en MongoDB Atlas, la plataforma en la nube de MongoDB.
3. Configuré un nuevo clúster:
 - Creé un nuevo clúster en MongoDB Atlas, ajustando la configuración según mis necesidades específicas.
4. Establecí la seguridad:
 - Configuré las medidas de seguridad necesarias para el clúster, incluyendo reglas de firewall y la creación de un usuario con los permisos adecuados.
5. Obtuve la URI de conexión:
 - Fui a la sección de "Connect" en MongoDB Atlas y obtuve la URI de conexión, que contenía información como el nombre de usuario, la contraseña, la dirección del servidor y el nombre de la base de datos.

Establecer el puerto

El puerto lo establecí en 5000 ya que no estaba en uso. Puede ver si su puerto esta en uso entrando a localhost:PORT y esperar que no este en uso.



Configuraciones iniciales en package.json

```
"main": "main.js",
```

Se modifica index.js por main.js por motivos de comprensión

```
"start": "nodemon app.js"
```

Se agrega el script para el arranque del servidor.

Creación de main.js (arranque del servidor)

main.js realiza la configuración inicial de la aplicación, conecta con la base de datos, establece middlewares para el procesamiento de solicitudes, gestión de sesiones y mensajes flash, y configura el motor de plantillas EJS. Además, inicia el servidor Express para escuchar las solicitudes entrantes.

1. Configuración de dotenv:

- **require("dotenv").config();** se utiliza para cargar variables de entorno desde un archivo **.env**. En este caso, es útil para almacenar información sensible, como la URL de conexión a la base de datos.

2. Conexión a la Base de Datos:

- **mongoose.connect(process.env.DB_URI, { useNewUrlParser: true, useUnifiedTopology: true });**
- Establece la conexión a la base de datos MongoDB utilizando la URL proporcionada en el archivo **.env**.
- **useNewUrlParser** y **useUnifiedTopology** son opciones para evitar advertencias y utilizar las últimas características de MongoDB.

3. Middlewares:

- **express.urlencoded** y **express.json** se utilizan para analizar datos de formularios en las solicitudes.
- **express.static('uploads')** sirve archivos estáticos desde el directorio **uploads**, lo que permite acceder a las imágenes cargadas.

4. Configuración de Sesiones:

- **express-session** se utiliza para gestionar sesiones en Express.
- Se configuran opciones como el secreto, si guardar sesiones no inicializadas y si forzar el almacenamiento de sesiones incluso si no han cambiado.

5. Middleware para Mensajes Flash:

- Crea un middleware que proporciona mensajes flash (mensajes temporales que se muestran al usuario y desaparecen después de una redirección).

6. Configuración del Motor de Plantillas EJS:

- `app.set("view engine", "ejs");` establece EJS como el motor de plantillas para la aplicación.

7. Prefijo de Rutas:

- `app.use("", require("./routes/routes"));` establece el prefijo de rutas para todas las rutas definidas en `routes.js`.

8. Inicio del Servidor:

- `app.listen(PORT, () => { console.log('Server corriendo en http://localhost:' + PORT); });` inicia el servidor en el puerto especificado o en el puerto 4000 por defecto.

Configuración de Express y Middleware

Configuración de Express:

En el archivo `main.js`, se importó el módulo `express` y se creó una instancia de la aplicación Express.

```
const express = require("express");
const app = express();
```

Configuración de Middlewares:

Se establecieron varios middlewares para habilitar funcionalidades esenciales en la aplicación.

```
// Middleware para procesar datos de formulario
app.use(express.urlencoded({ extended: false }));
app.use(express.json());

// Middleware para servir archivos estáticos desde el directorio 'uploads'
app.use(express.static('uploads'));

// Middleware para gestionar sesiones de usuario
app.use(session({
  secret: '666',
  saveUninitialized: true,
  resave: false,
}));
```


Procesamiento de Datos de Formulario:

express.urlencoded({ extended: false }): Configura Express para analizar datos de formulario codificados en la URL.

express.json(): Habilita el análisis de datos JSON en las solicitudes.

Servir Archivos Estáticos:

express.static('uploads'): Permite que Express sirva archivos estáticos desde el directorio 'uploads'. Esto es especialmente útil para servir imágenes cargadas por los usuarios.

Gestión de Sesiones:

express-session: Middleware que facilita la gestión de sesiones de usuario en Express. Se utiliza para crear y gestionar la sesión de usuario.

Configuración del Motor de Plantillas EJS:

Se configuró EJS como el motor de plantillas para generar vistas HTML dinámicas.

```
app.set("view engine", "ejs");
```

Establece EJS como el motor de plantillas para la aplicación Express.

Configuración del Puerto y Escucha del Servidor:

Se configuró el puerto en el que la aplicación Express escuchará las solicitudes y se inició el servidor:

```
const PORT = process.env.PORT || 4000;  
app.listen(PORT, () => {  
  console.log('Server corriendo en http://localhost:' + PORT);  
});
```

process.env.PORT || 4000: Configura el puerto en el que se ejecutará el servidor. Si la variable de entorno PORT está definida, se utiliza; de lo contrario, se utiliza el puerto 4000.

app.listen(PORT, () => { /* ... */ }): Inicia el servidor Express y escucha las solicitudes en el puerto especificado.

Creación de programs.js (Modelo de Mongoose)

El archivo programs.js es un modelo de Mongoose que define la estructura y el comportamiento de los documentos que se almacenarán en la colección de programas en la base de datos MongoDB.

```
// Importa la biblioteca mongoose
const mongoose = require("mongoose");

// Define el esquema del programa utilizando la clase Schema de mongoose
const programSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
  image: {
    type: String,
    default: "",
  },
});

// Crea y exporta el modelo Program utilizando el esquema definido
module.exports = mongoose.model("Program", programSchema);
```

Importación de Mongoose:

Se importa la biblioteca Mongoose, que proporciona una interfaz para interactuar con MongoDB desde Node.js.

Definición del Esquema:

Se define el esquema del programa utilizando la clase Schema de Mongoose.

El esquema especifica los campos que tendrán los documentos en la colección de programas.

El esquema tiene campos como **name**, **description**, **price**, y **image**. Cada campo tiene un tipo y, en algunos casos, restricciones como la propiedad **required**.

Creación del Modelo:

```
module.exports = mongoose.model("Program", programSchema);
```

Se crea y exporta el modelo **Program** utilizando el esquema definido.

Un modelo es una instancia de un esquema, y se utiliza para interactuar con la colección asociada en la base de datos.

Funcionalidades del Modelo:

name, description, y price: Son campos obligatorios que representan el nombre, descripción y precio del programa, respectivamente.

image: Es un campo opcional que representa el nombre del archivo de imagen asociado al programa.

default: "" (en image): Si no se proporciona una imagen al agregar un programa, el campo se establece en una cadena vacía por defecto.

Configuración de Rutas

Importación de Módulos y Modelos:

Se importaron los módulos y modelos necesarios, incluyendo Express, el enrutador de Express y el modelo de mongoose para los programas.

```
const express = require("express");
const router = express.Router();
const Program = require('../models/programs');
```

Configuración de Multer para Manejo de Imágenes:

Se configuró **multer** para gestionar el almacenamiento de imágenes en el servidor.

```
const multer = require('multer');
var storage = multer.diskStorage({
  /* ... */
});
var upload = multer({
  storage: storage,
}).single("image");
```

multer.diskStorage({ /* ... */ }): Configuración del almacenamiento en disco para las imágenes.

multer({ storage: storage }).single("image"): Creación de un middleware de multer para manejar la carga de una sola imagen con el nombre "image".

Ruta para Agregar un Programa:

Se definió una ruta para manejar la solicitud POST para agregar un nuevo programa.

```
router.post('/agregar', upload, async (req, res) => { /* ... */ });
```

Se utilizó el middleware de **multer** para procesar la carga de imágenes y se guardó la información del programa en la base de datos.

Ruta para Obtener Todos los Programas:

Se definió una ruta para manejar la solicitud GET para obtener todos los programas.

```
router.get("/", async (req, res) => { /* ... */ });
```

Se utilizó el modelo de mongoose para encontrar todos los programas en la base de datos y se renderizó la página principal con la información obtenida.

Rutas para Editar y Actualizar un Programa

Se crearon rutas para manejar la edición y actualización de programas.

```
router.get("/edit/:id", async (req, res) => { /* ... */ });  
  
router.post('/actualizar/:id', upload, async (req, res) => { /* ... */ });
```

- La ruta de edición recuperó la información del programa según su ID y renderizó la página de edición.
- La ruta de actualización procesó la solicitud POST para actualizar la información del programa en la base de datos.

Ruta para Eliminar un Programa:

Se creó una ruta para manejar la eliminación de programas.

```
router.get('/delete/:id', async (req, res) => { /* ... */ });
```

Se eliminó el programa de la base de datos y se intentó eliminar la imagen asociada del sistema de archivos.

Rutas para Renderizar Páginas Adicionales

```
router.get("/agregar", (req, res) => { /* ... */ });
```

```
router.get("/info", (req, res) => { /* ... */ });
```

Estas rutas simplemente renderizaron las plantillas correspondientes sin realizar operaciones CRUD.

Exportación de Rutas Configuradas

```
module.exports = router;
```

Todas las rutas configuradas se exportaron al final del archivo para su uso en el archivo principal (**main.js**).

Creación y configuración de las vistas EJS (Embedded JavaScript)

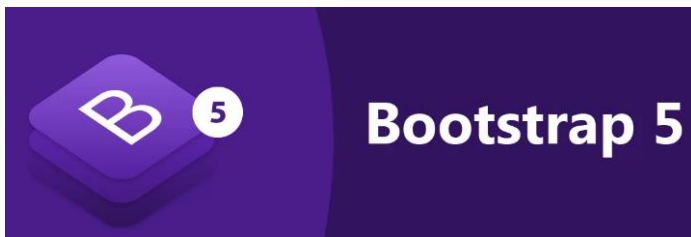
Estructura de directorios: En el directorio **views**, organicé mis archivos EJS para reflejar las diferentes secciones y páginas de mi aplicación. Cada vista tiene un propósito específico y se encarga de mostrar información relacionada con las operaciones CRUD y otras funciones.

- **index.ejs:** Esta vista representa la página principal de mi aplicación. Utilicé DataTables para crear una tabla interactiva que muestra los programas almacenados en la base de datos. Integré elementos visuales y de diseño para mejorar la experiencia del usuario.
- **add_programs.ejs:** Diseñé esta vista para el formulario de agregar programas. Incluí campos para el nombre, descripción, precio e imagen del programa. La integración con **multer** facilitó la carga de imágenes.
- **edit_programs.ejs:** Esta vista permite la edición de programas existentes. Implementé un formulario similar al de agregar programas, pero con los campos ya llenos con la información actual del programa seleccionado.
- **about.ejs:** En esta vista, proporcioné información sobre la aplicación y su propósito. Es una sección informativa que complementa la funcionalidad principal.

Uso de EJS para dinamismo: Dentro los archivos EJS, aproveché la capacidad de EJS para incrustar código JavaScript en HTML. Esto permitió generar contenido dinámico y presentar los datos de los programas de manera estructurada en las páginas.

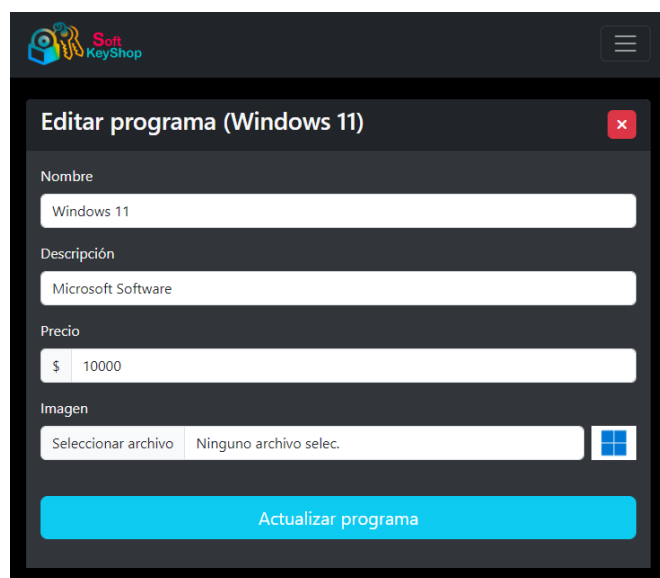
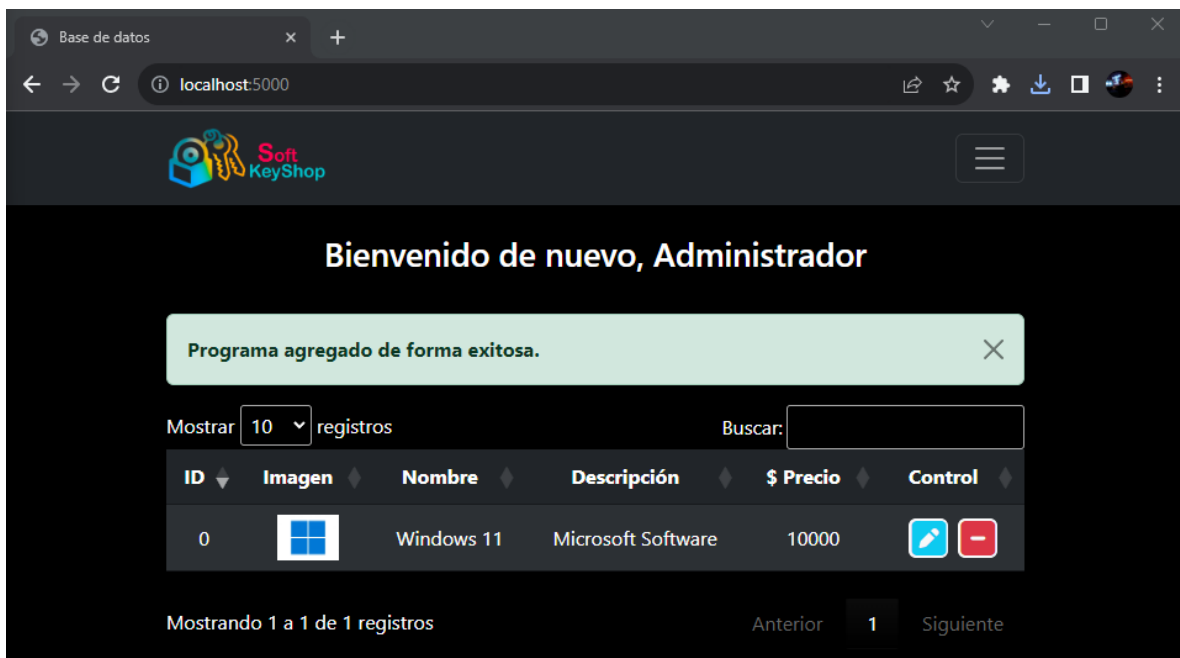
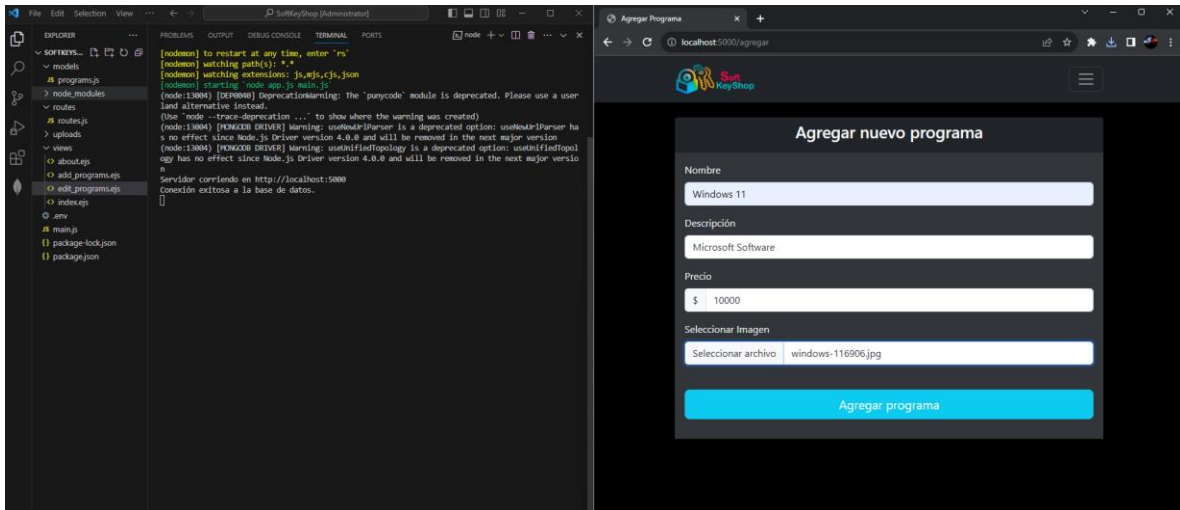
```
<% programs.forEach((row, index) => { %>
  ...
  <td class="text-center"><%= row.name %></td>
  ...
<% }) %>
```

Integración de FontAwesome, Bootstrap y DataTables: Para mejorar la apariencia y la usabilidad de mis vistas, incorporé estilos de Bootstrap y utilicé DataTables para crear tablas interactivas. Esto facilitó la visualización y manipulación de datos tabulares. Implementé el manejo de mensajes flash en mis vistas para proporcionar retroalimentación al usuario después de realizar ciertas operaciones, como agregar, editar o eliminar programas.



Pruebas y Depuración

Se realizaron pruebas de las rutas y se depuraron errores.



Agradecimientos:

Esta aplicación web no hubiera sido posible gracias a la profesora **Margarita Mondragon Arellano** y el tutor de **DCodeMania** de <https://dcodemania.com/> .