**Documentación de Pedidos&Entregas**

**Planteamiento del problema:**

La compañía Pedidos&Entregas lo ha contratado a usted para diseñar el sistema que le

permitirá realizar la toma de pedidos por empleado, nomina completa del personal,

activar o inactivar empleados, este proceso le permitirá a la empresa agilizar los procesos

operativos y logísticos.

Pedidos&Entregas cuenta con un personal de 100 empleados distribuidos de la siguiente

manera

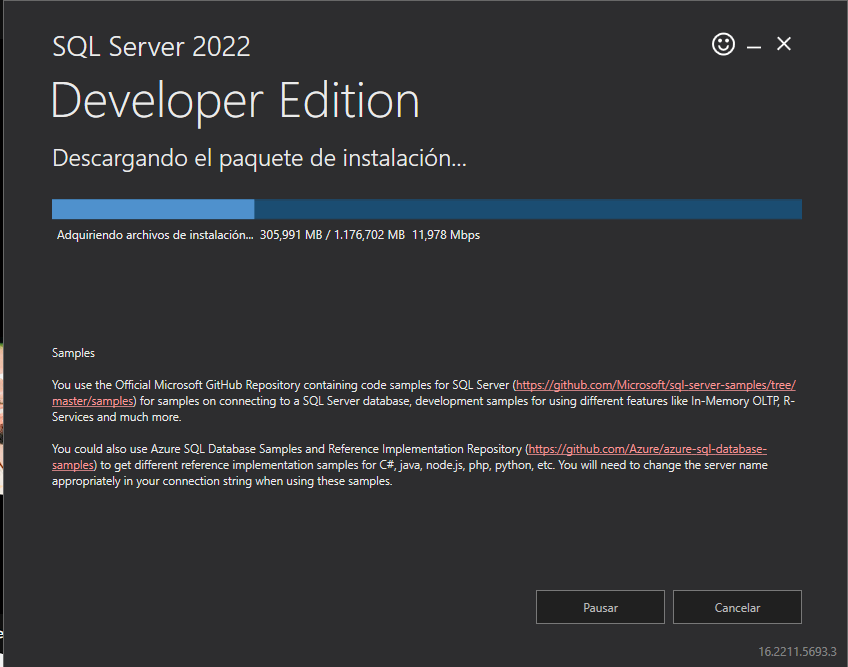
* 70 empleados para toma de pedidos
* 20 empleados administrativos
* 8 empleados de entrega
* 2 gerentes (administradores)

***Las herramientas usadas para este programa son:***

SQL server, nodeJs (Back), react (front)

**Base de datos**

*La base de datos estará creada en SQL serve lo cual tendrá mucha versatilidad para este proyecto*



*Se instala ssms para tener una interfaz gráfica para poder modelar*

SQL Server Management Studio (SSMS) es un entorno integrado para administrar cualquier infraestructura SQL, desde SQL Server hasta Azure SQL Database. SSMS proporciona herramientas para configurar, monitorear y administrar instancias de SQL Server y bases de datos. Utilice SSMS para implementar, monitorear y actualizar los componentes de la capa de datos utilizados por sus aplicaciones y crear consultas y scripts.SQL Server Management Studio (SSMS) es un entorno integrado para administrar cualquier infraestructura SQL, desde SQL Server hasta Azure SQL Database. SSMS proporciona herramientas para configurar, monitorear y administrar instancias de SQL Server y bases de datos. Utilice SSMS para implementar, monitorear y actualizar los componentes de la capa de datos utilizados por sus aplicaciones y crear consultas y scripts.SQL Server Management Studio (SSMS) es un entorno integrado para administrar cualquier infraestructura SQL, desde SQL Server hasta Azure SQL Database. SSMS proporciona herramientas para configurar, monitorear y administrar instancias de SQL Server y bases de datos. Utilice SSMS para implementar, monitorear y actualizar los componentes de la capa de datos utilizados por sus aplicaciones y crear consultas y scripts.

***Script de la base de datos:***

**-- Crear la base de datos**

CREATE DATABASE DB\_Pedidos\_entrega;

GO

-- Usar la base de datos

USE DB\_Pedidos\_entrega;

GO

**-- Tabla Empleado**

CREATE TABLE Empleado (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del empleado

Nombre VARCHAR(255), -- Nombre del empleado

Apellidos VARCHAR(255), -- Apellidos del empleado

Cedula VARCHAR(20), -- Número de cédula del empleado

Direccion VARCHAR(255), -- Dirección del empleado

Departamento VARCHAR(50), -- Departamento donde trabaja el empleado

Municipio VARCHAR(50), -- Municipio donde trabaja el empleado

Barrio VARCHAR(50), -- Barrio donde trabaja el empleado

TipoEmpleadoId INT, -- ID del tipo de empleado al que pertenece

Salario DECIMAL(10, 2), -- Salario del empleado

FOREIGN KEY (TipoEmpleadoId) REFERENCES TipoEmpleado(Id) -- Relación con la tabla TipoEmpleado

);

**-- Tabla TipoEmpleado**

CREATE TABLE TipoEmpleado (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del tipo de empleado

Descripcion VARCHAR(255), -- Descripción del tipo de empleado

Codigo VARCHAR(20) -- Código del tipo de empleado

);

**-- Tabla Cliente**

CREATE TABLE Cliente (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del cliente

Nombre VARCHAR(255), -- Nombre del cliente

Apellidos VARCHAR(255), -- Apellidos del cliente

Cedula VARCHAR(20), -- Número de cédula del cliente

Direccion VARCHAR(255), -- Dirección del cliente

Departamento VARCHAR(50), -- Departamento del cliente

Municipio VARCHAR(50), -- Municipio del cliente

Barrio VARCHAR(50), -- Barrio del cliente

Activo BIT, -- Indicador de si el cliente está activo o no

FechaNacimiento DATE, -- Fecha de nacimiento del cliente

Telefono VARCHAR(20) -- Número de teléfono del cliente

);

**-- Tabla Categorías**

CREATE TABLE Categorias (

Id INT IDENTITY(1,1)PRIMARY KEY,

Nombre VARCHAR(100)

);

**-- Tabla Marcas**

CREATE TABLE Marcas (

Id INT IDENTITY(1,1)PRIMARY KEY,

Nombre VARCHAR(100)

);

**-- Tabla Producto**

CREATE TABLE Producto (

Id INTIDENTITY(1,1)PRIMARY KEY, -- Identificador único del producto

Codigo VARCHAR(20), -- Código del producto

Nombre VARCHAR(255), -- Nombre del producto

Descripcion TEXT, -- Descripción detallada del producto

Marca VARCHAR(100), -- Marca del producto

Categoria VARCHAR(100), -- Categoría del producto

CantidadDisponible INT, -- Cantidad disponible en inventario

Habilitado BIT, -- Indicador de si el producto está habilitado o no

CantidadVentas INT, -- Cantidad de ventas realizadas del producto

Iva DECIMAL(5, 2), -- Tasa de impuesto al valor agregado

Precio DECIMAL(10, 2) -- Precio del producto

);

**-- Tabla de Relación entre Marcas y Productos**

CREATE TABLE MarcaProducto (

MarcaId INT,

ProductoId INT,

PRIMARY KEY (MarcaId, ProductoId),

FOREIGN KEY (MarcaId) REFERENCES Marcas(Id),

FOREIGN KEY (ProductoId) REFERENCES Producto(Id)

);

**-- Tabla de Relación entre Categorías y Productos**

CREATE TABLE CategoriaProducto (

CategoriaId INT,

ProductoId INT,

PRIMARY KEY (CategoriaId, ProductoId),

FOREIGN KEY (CategoriaId) REFERENCES Categorias(Id),

FOREIGN KEY (ProductoId) REFERENCES Producto(Id)

);

**-- Tabla Carrito**

CREATE TABLE Carrito (

Id INT IDENTITY(1,1) PRIMARY KEY,

ClienteId INT,

FechaCreacion DATE,

FOREIGN KEY (ClienteId) REFERENCES Cliente(Id)

);

**-- Tabla DetalleCarrito (para la relación muchos a muchos entre Carrito y Producto)**

CREATE TABLE DetalleCarrito (

Id INT IDENTITY(1,1)PRIMARY KEY,

CarritoId INT,

ProductoId INT,

Cantidad INT,

FOREIGN KEY (CarritoId) REFERENCES Carrito(Id),

FOREIGN KEY (ProductoId) REFERENCES Producto(Id)

);

**-- Tabla Pedido**

CREATE TABLE Pedido (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del pedido

NumeroOrden VARCHAR(20), -- Número de orden del pedido

FechaRegistro DATE, -- Fecha de registro del pedido

TotalPrecio DECIMAL(10, 2), -- Total de precio de los productos en el pedido

TotalIva DECIMAL(10, 2), -- Total del impuesto al valor agregado en el pedido

Departamento VARCHAR(50), -- Departamento de entrega del pedido

Ciudad VARCHAR(50), -- Ciudad de entrega del pedido

Barrio VARCHAR(50), -- Barrio de entrega del pedido

DireccionEntrega VARCHAR(255), -- Dirección de entrega del pedido

ClienteId INT, -- ID del cliente que realizó el pedido

EmpleadoId INT, -- ID del empleado que tomó el pedido

EntregadorId INT, -- ID del empleado que entregó el pedido

Observacion TEXT, -- Observaciones adicionales sobre el pedido

);

**-- Tabla DetallePedido**

CREATE TABLE DetallePedido (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del detalle del pedido

PedidoId INT, -- ID del pedido al que pertenece el detalle

NombreProducto VARCHAR(255), -- Nombre del producto en el detalle

DescripcionProducto TEXT, -- Descripción detallada del producto en el detalle

Precio DECIMAL(10, 2), -- Precio unitario del producto en el detalle

Iva DECIMAL(5, 2), -- Tasa de impuesto al valor agregado del producto en el detalle

Cantidad INT, -- Cantidad del producto en el detalle

CodigoProducto VARCHAR(20), -- Código del producto en el detalle

Marca VARCHAR(100), -- Marca del producto en el detalle

Categoria VARCHAR(100), -- Categoría del producto en el detalle

FOREIGN KEY (PedidoId) REFERENCES Pedido(Id) -- Relación con la tabla Pedido

);

**-- Tabla EmpleadoPedido (Relación Muchos a Muchos entre Empleado y Pedido)**

CREATE TABLE EmpleadoPedido (

EmpleadoId INT,

PedidoId INT,

PRIMARY KEY (EmpleadoId, PedidoId),

FOREIGN KEY (EmpleadoId) REFERENCES Empleado(Id),

FOREIGN KEY (PedidoId) REFERENCES Pedido(Id)

);

**-- Tabla ClientePedido (Relación Muchos a Muchos entre Cliente y Pedido)**

CREATE TABLE ClientePedido (

ClienteId INT,

PedidoId INT,

PRIMARY KEY (ClienteId, PedidoId),

FOREIGN KEY (ClienteId) REFERENCES Cliente(Id),

FOREIGN KEY (PedidoId) REFERENCES Pedido(Id)

);

**-- Tabla DetalleProducto (para la relación muchos a muchos entre DetallePedido y Producto)**

CREATE TABLE DetalleProducto (

DetalleId INT,

ProductoId INT,

PRIMARY KEY (DetalleId, ProductoId),

FOREIGN KEY (DetalleId) REFERENCES DetallePedido(Id),

FOREIGN KEY (ProductoId) REFERENCES Producto(Id)

);

**-- Tabla Usuario**

CREATE TABLE Usuario (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del usuario

EmpleadoId INT, -- ID del empleado asociado al usuario

Usuario VARCHAR(50), -- Nombre de usuario

Contraseña VARCHAR(255), -- Contraseña encriptada del usuario

FechaRegistro DATE, -- Fecha de registro del usuario

FOREIGN KEY (EmpleadoId) REFERENCES Empleado(Id) -- Relación con la tabla Empleado

);

**-- Tabla Departamento**

CREATE TABLE Departamento (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del departamento

Nombre VARCHAR(100) -- Nombre del departamento

);

**-- Tabla Municipio**

CREATE TABLE Municipio (

Id INT IDENTITY(1,1)PRIMARY KEY, -- Identificador único del municipio

Nombre VARCHAR(100), -- Nombre del municipio

DepartamentoId INT, -- ID del departamento al que pertenece el municipio

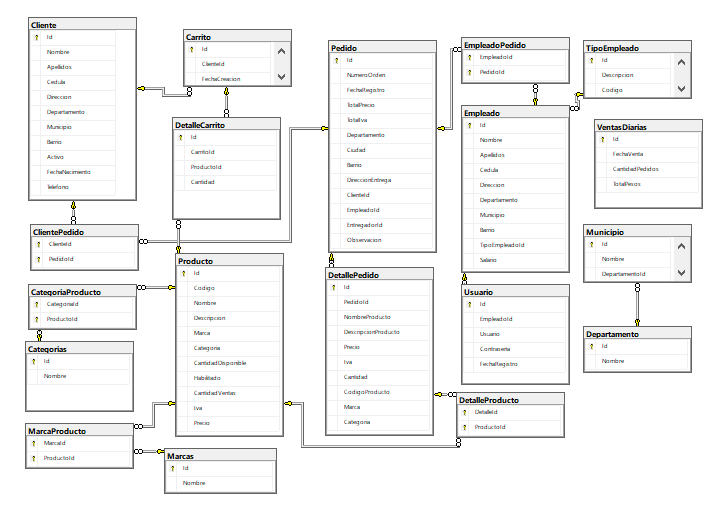
FOREIGN KEY (DepartamentoId) REFERENCES Departamento(Id) -- Relación con la tabla Departamento

);

***Se agregaron las siguientes tablas adicionales***

* Marca – MarcaProducto *(relación de muchos a muchos)*
* Categoría – CategoriaProducto *(relación de muchos a muchos)*
* Carrito – DetalleCarrito (*relación de muchos a muchos)*
* DetallePedido
* Ventas diarias
* empleadoPedido (*relación de muchos a muchos)*
* clientePedido *(relación de muchos a muchos)*

**Diagrama de base de datos llamada DB\_Pedidos\_entrega**

****

**Back - end (NodeJs)**

*SE USA LA HERRAMIENTA DE EDICION VISUAL STUDIO CODE*

***Se inicia con npm init -y en la terminal***

***Se ejecuta el código con npm run dev***

**Se instala las siguientes librerías**

{

  "description": "",

  "dependencies": {

    "cors": "^2.8.5",

    "dotenv": "^16.3.1",

    "express": "^4.18.2",

    "morgan": "^1.10.0",

    "mssql": "^10.0.1"

  }

}

* **cors**: Un middleware de Express js que se utiliza para habilitar la política de mismo origen en aplicaciones web. La versión especificada es ^2.8.5", lo que significa que se permite cualquier versión compatible con la versión 28.5 o superior, pero no una versión 3.0 o posterior.
* **dotenv**: Un paquete que facilita la carga de variables de entorno desde un archivo .env en tu proyecto. La versión especificada es 16.3.1.
* **express**: El framework web de Nodejs ampliamente utilizado. La versión especificada es ^4.18.5.
* **morgan**: Un middleware de registro de solicitudes HT TP para Express js. La versión especificada es "^1.10.0".

• **mssql**: Un paquete que permite interactuar con bases de datos SOL Server desde una aplicación Node js. La versión especificada es ^10.0.1.

**Se instala las siguientes librerías para poder tener ES6 (código moderno)**

  "devDependencies": {

    "@babel/cli": "^7.22.15",

    "@babel/core": "^7.22.20",

    "@babel/node": "^7.22.19",

    "@babel/preset-env": "^7.22.20",

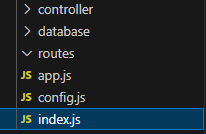
    "nodemon": "^3.0.1"

  }

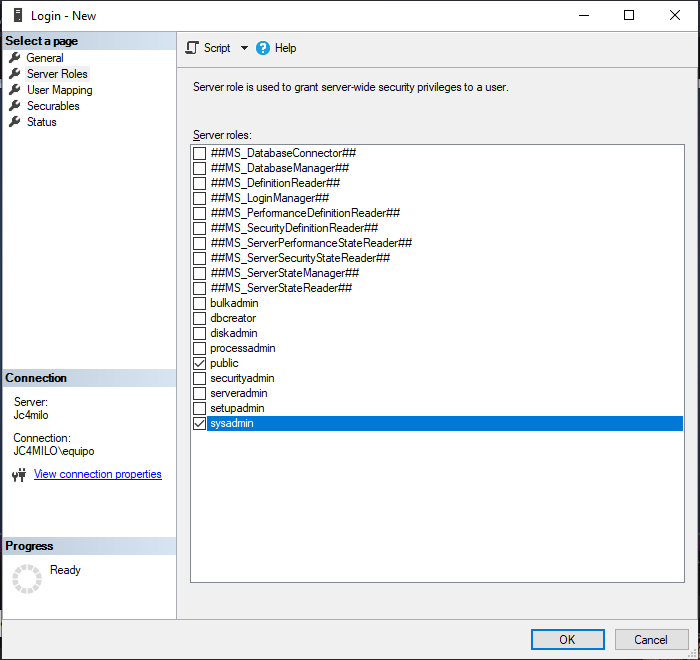
* **@babel/core**: Babel es una herramienta ampliamente utilizada para transpilar (convertir) código JavaScript moderno (como ES6/ES7) en una versión anterior que sea compatible con navegadores y entornos más antiguos.
* **@babel/core:** es el núcleo de Babel y proporciona la funcionalidad principal para la transpilación del código.
* **@babel/cli**:@babel/cli es una interfaz de línea de comandos para Babel. Permite que Babel sea utilizado desde la línea de comandos para compilar archivos JavaScript.
* **@babel/preset-env**: Los presets en Babel son conjuntos predefinidos de reglas de transpilación que permiten a los desarrolladores especificar qué características de JavaScript moderno desean admitir en sus proyectos.
* **@babel/preset-env:** es un preset que se utiliza para determinar automáticamente las transformaciones de código necesarias en función del entorno objetivo (por ejemplo, los navegadores que se pretende admitir).
* **@babel/node**: @babel/node es un paquete que permite ejecutar scripts de Node.js con el transpilador Babel aplicado. Esto es útil para ejecutar código JavaScript moderno directamente en Node.js, incluso antes de la transpilación.
* **nodemon**: **nodemon** es una herramienta de desarrollo que supervisa cambios en los archivos de tu proyecto y automáticamente reinicia la aplicación cuando detecta un cambio. Esto es útil durante el desarrollo para evitar tener que reiniciar manualmente la aplicación cada vez que se hace un cambio en el código.

En conjunto, estas dependencias son comunes en proyectos de Node.js que utilizan Babel para admitir código **JavaScript** moderno y **nodemon** para facilitar el desarrollo continuo sin interrupciones. Además, @babel/preset-env ayuda a asegurarse de que el código transpilado sea compatible con los navegadores y entornos objetivo específicos.

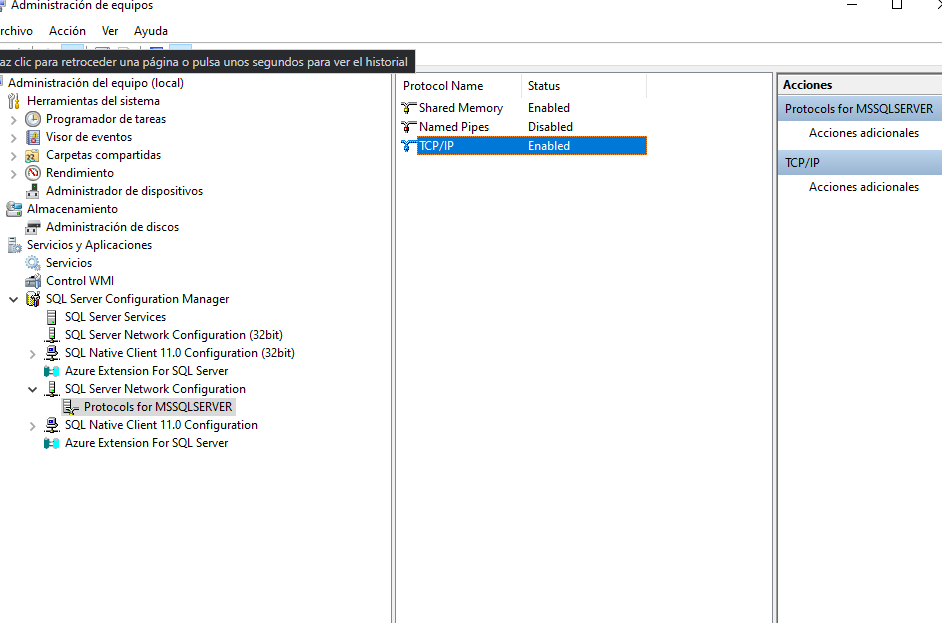
*Se crean las siguientes carpetas y archivos para poder ejecutar el código*

****

1. Para poder hacer la conexión se debe tener el usuario y contraseña en lugar de la autenticación de Windows ya que se necesita ingresar a los datos y que el **back end** lo pueda leer correctamente, se debe dar el rol de admin para poder darle funcionalidades

****

1. Nececitas el TCP/IP ACTIVADO y ya que puedes dar conflictos a la hora de hacer una *consulta*

****

1. Importa la base de datos ya creada llamada dbSettings trayendo el usuario de la base de datos Se hace la conexión async para hacer una conexión a través de los atributos ya puestos en dbSetting, con esa conexión se le hace una petición a la base de datos (SELECT 1) Y lo mostremos en consola

import sql from 'mssql';

const dbSettings = ({

    user: "admin",

    passowrd: "admin1234",

    server: "Localhost",

    databasse: "DB\_Pedidos\_entregas"

})

const webConection = async () => {

    const pool = await sql.connect(dbSettings)

    const result = await pool.request().query('SELECT 1')

}

webConection();

Esta es la función más sencilla en ciertos aspectos, ya que solo se debe mostrar lo que hay en la base de datos

//trae todos los produsctos

export const getProducts = async (req, res) => {

    try {

        const pool = await webConnection();

        const result = await pool.request().query(query.getProducts);

        //api products

        res.json(result.recordset);

    } catch (error) {

        console.error('Error al ejecutar la consulta SQL:', error);

        res.status(500).json('Error en la consulta SQL');

    }

}

Crear nuevo producto trayendo todos los atributos en un formato json

Se debe primero la expresión express para que pueda interpretar el código que le estamos enviando a la base de datos

//middleware

app.use(express.json());

// esto nos ayudara a leer código del html o del formulario el cual le estamos enviando los datos

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

app.use(productsRoutes);

se debe colocar a cada uno los tipos de valores que se están registrando en la base de datos para que no haya conflictos

// crea un nuevo producto

export const CreateProducts = async (req, res) => {

    let { id, Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio } = req.body

    const pool = await webConnection();

    await pool

        .request()

        .input("id", sql.Int, id)

        .input("Codigo", sql.VarChar, Codigo)

        .input("Nombre", sql.VarChar, Nombre)

        .input("Descripcion", sql.Text, Descripcion)

        .input("CantidadDisponible", sql.Int, CantidadDisponible)

        .input("Habilitado", sql.Bit, Habilitado)

        .input("CantidadVentas", sql.Int, CantidadVentas)

        .input("Iva", sql.Decimal, Iva)

        .input("Precio", sql.Decimal, Precio)

        .query("INSERT INTO Producto (id, Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio ) VALUES (@id, @Codigo, @Nombre, @Descripcion, @CantidadDisponible, @Habilitado, @CantidadVentas, @Iva, @Precio )");

    res.json('new product')

}

Una manera mas facil de no tener esta línea de código

.query("INSERT INTO Producto (id, Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio ) VALUES (@id, @Codigo, @Nombre, @Descripcion, @CantidadDisponible, @Habilitado, @CantidadVentas, @Iva, @Precio )");

Puedes crear un archivo separado para tener todas las consultas, en la carpeta database se crea el archivo query es el que tendrá todas las consultas se debe tener en cuenta que debe ser en formato de objeto para poder tener varias y poder usarlas importándolas y llamando des la siguiente manera .query(query.UpdateProduct);

export default {

    getProducts: 'SELECT \* FROM Producto',

    createNewProduct: "INSERT INTO Producto (id, Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio ) VALUES (@id, @Codigo, @Nombre, @Descripcion, @CantidadDisponible, @Habilitado, @CantidadVentas, @Iva, @Precio )",

    IdProduct: "SELECT \* FROM Producto WHERE Id = @id",

    DeleteProduct: "DELETE FROM Producto WHERE Id = @id",

    CountProduct: "SELECT COUNT(\*) FROM Producto",

    UpdateProduct: "UPDATE Producto SET  Codigo = @Codigo, Nombre = @Nombre, Descripcion = @Descripcion, CantidadDisponible = @CantidadDisponible, Habilitado = @Habilitado,  CantidadVentas = @CantidadVentas, Iva = @Iva, Precio = @Precio WHERE Id = @id"

}

La Función de eliminar es muy esencial al crear el código, se usa el id que tiene el objeto para poder eliminar cierto elemento

// elimina un solo producto

export const DeleteProductId = async (req, res) => {

    const { id } = req.params;

    const pool = await webConnection();

    try {

        const result = await pool

            .request()

            .input("Id", id)

            .query(query.DeleteProduct)

        res.send("succes", 204)

    } catch (error) {

        console.error('Error al ejecutar la consulta SQL:', error);

        res.status(500).json('Error en el post SQL');

    }

}

La función de actualizar es similar en ciertos aspectos se parece a la de crear y tambien a la de eliminar ya que se tiene que esta en constante cambio

// actualizar cada producto

export const UpdateProduct = async (req, res) => {

    let { Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio } = req.body

    const { id } = req.params;

    const pool = await webConnection();

    await pool

        .request()

        .input("Id", sql.Int, id)

        .input("Codigo", sql.VarChar, Codigo)

        .input("Nombre", sql.VarChar, Nombre)

        .input("Descripcion", sql.Text, Descripcion)

        .input("CantidadDisponible", sql.Int, CantidadDisponible)

        .input("Habilitado", sql.Bit, Habilitado)

        .input("CantidadVentas", sql.Int, CantidadVentas)

        .input("Iva", sql.Decimal, Iva)

        .input("Precio", sql.Decimal, Precio)

        .query(query.UpdateProduct);

    res.json( Codigo, Nombre, Descripcion, CantidadDisponible, Habilitado, CantidadVentas, Iva, Precio );

Esta es una función extra que nos ayudara a poder contar los productos de una manera mucho mas eficaz

// contar los productos

export const CountController = async (req, res) => {

    const pool = await webConnection();

    try {

        const result = await pool

            .request()

            .query(query.CountProduct)

        res.json(result.recordset[0][''])

    } catch (error) {

        console.error('Error al ejecutar la consulta SQL:', error);

        res.status(500).json('Error en el post SQL');

    }

}

Con estos pasos creamos las siguiente rutas y controladores para poder tener una api mas completa y pueda consumirse de una mejor manera en el front

**Front - End (React)**