**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) Git

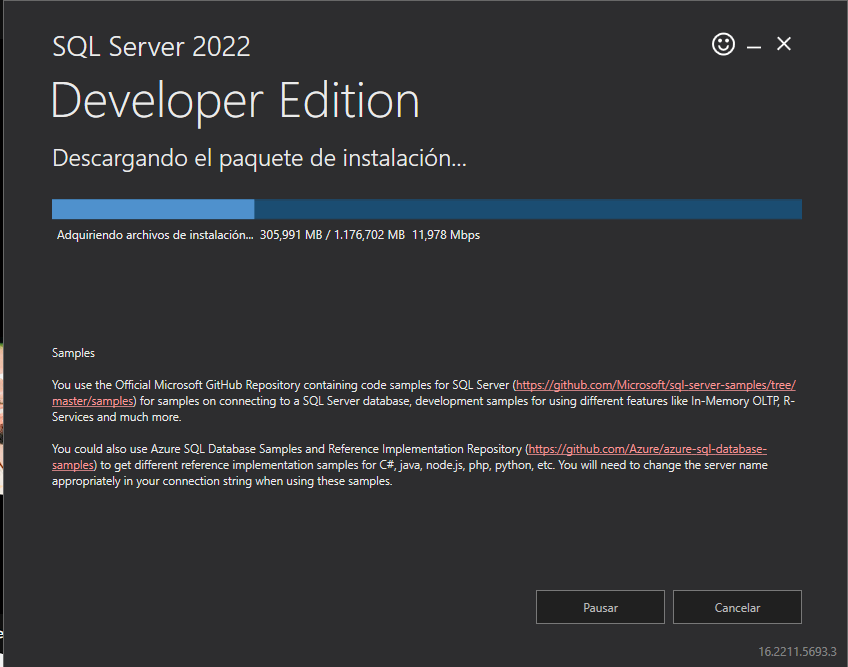
Se crea el repositorio Pedidos\_Entregas se crea la rama Development para hacer todos los cambios

**Para instalar todos los recursos del back y del front con el comando npm i en las terminales de front y el back**

***El aplicativo web se realizó de forma Local***

**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

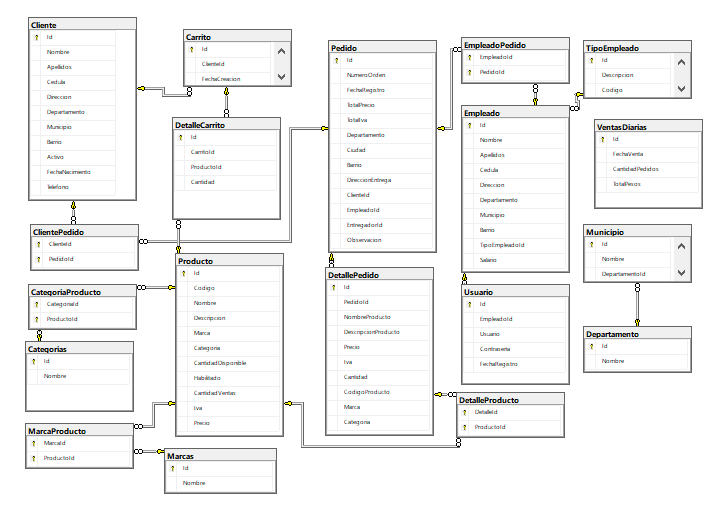
FOREIGN KEY (DepartamentoId) REFERENCES Departamento(Id) -- Relación con la tabla Departamento

);

***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",

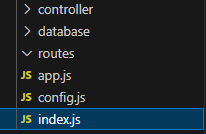
    "nodemon": "^3.0.1"

  }

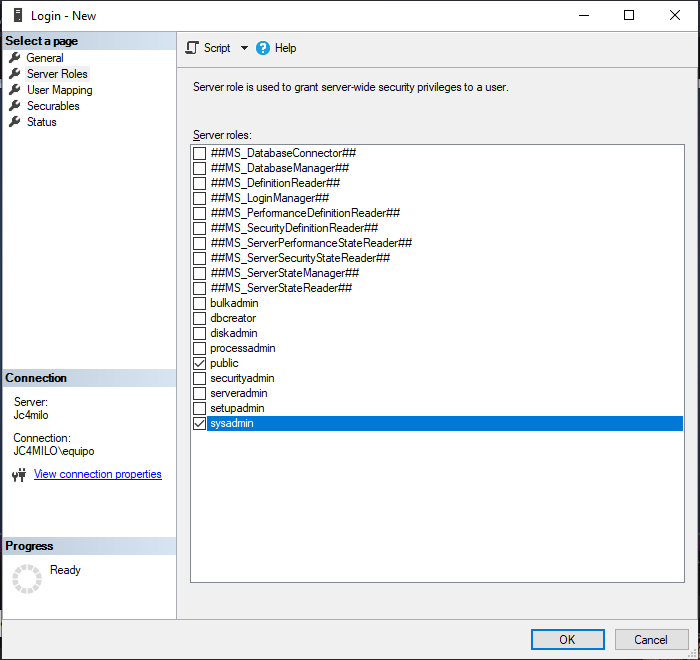
* **@babel/core**: Babel es una herramienta ampliamente utilizada para transpilar (convertir) código JavaScript moderno (como ES6/ES7) en una versión anterior que sea compatible con navegadores y entornos más antiguos.
* **@babel/core:** es el núcleo de Babel y proporciona la funcionalidad principal 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.

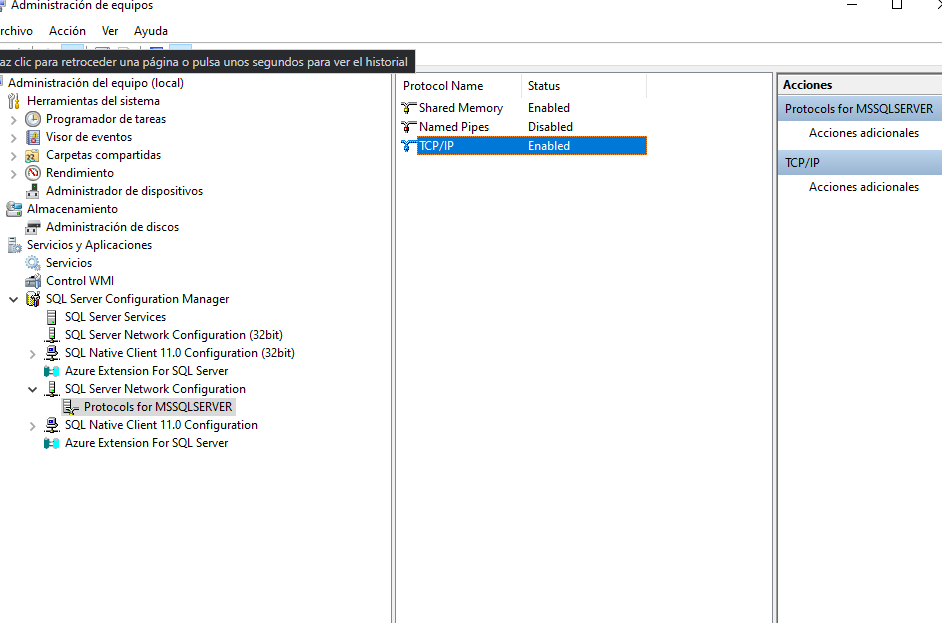
*Se crean las siguientes carpetas y archivos para poder ejecutar el código*

****

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

**JWT**

Para por tener JWT en la api Se necesita hacer el siguiente comando en la consola o terminal de visual studio code:

*npm install jsonwebtoken*

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

    res.send('welcome to my Api Jc4milo © Developer 2023');

})

app.post('/api/login', (req, res) => {

    const user = {

        name: "admin",

        correo: "admin@gmail.com",

        contraseña: "admin12345"

    };

    jwt.sign({ user }, 'secretKey', { expiresIn: '180s' }, (error, token) => {

        if (error) {

            res.status(500).json({ mensaje: 'Error al generar el token' });

        } else {

            res.json({ token });

        }

    });

});

Esta ruta está configurada para manejar solicitudes HTTP GET en la URL '/api'.

Cuando alguien accede a esta ruta en el navegador o mediante una solicitud GET, la aplicación responderá con un mensaje de bienvenida.

El mensaje que se envía en la respuesta es: "welcome to my Api Jc4milo © Developer 2023".

En resumen, esta ruta sirve como una página de bienvenida para la API y muestra un mensaje al usuario cuando accede a ella.

Ruta POST '/api/login':

Esta ruta está configurada para manejar solicitudes HTTP POST en la URL '/api/login'.

Cuando alguien realiza una solicitud POST a esta ruta, la aplicación genera un token JWT (JSON Web Token).

El token se genera utilizando la biblioteca jsonwebtoken y se firma con una "clave secreta" llamada 'secretKey'.

En este caso, se crea un objeto user con un nombre de usuario, correo electrónico y contraseña simulados.

Luego, se firma el token utilizando el objeto user como datos de carga útil (payload) y se configura para que expire en 180 segundos (3 minutos).

Si se genera con éxito el token, la aplicación responde con un objeto JSON que contiene el token.

Si hay un error al generar el token, la aplicación responde con un mensaje de error y un estado HTTP 500 (Error interno del servidor).

Esta ruta generalmente se utiliza para autenticar a los usuarios y proporcionarles un token de acceso después de que ingresen sus credenciales.

En resumen, estas dos rutas son parte de una aplicación web que proporciona una página de bienvenida en la ruta GET '/api' y permite a los usuarios autenticarse y obtener un token JWT en la ruta POST '/api/login'. El token JWT puede ser utilizado para autenticación en otras partes de la aplicación.

La función verifyToken es como el guardia de seguridad de tu aplicación web. Su trabajo es asegurarse de que las personas que intentan acceder a ciertas partes de tu sitio estén realmente autorizadas.

Primero, revisa la solicitud que llega y busca un "token" en el encabezado de autorización. Este token es como una tarjeta de acceso especial que debería venir con la solicitud.

Si encuentra ese token y está en el formato correcto ("Bearer <token>"), lo toma y verifica si es válido. Utiliza una "clave secreta" llamada 'secretKey' para hacer esta verificación.

Si el token es válido, significa que el usuario está autenticado. Entonces, guarda la información de autenticación en la solicitud para que puedas usarla más adelante en las rutas protegidas y permite que la solicitud continúe su camino.

Pero si el token no es válido, o si no se proporciona ningún token, entonces esta función le dice al navegador "¡Alto! No puedes pasar" y envía un mensaje de error con un código de estado HTTP 403 (prohibido) para decirle al usuario que algo salió mal.

Así que, en pocas palabras, esta función se encarga de la seguridad de tu aplicación web, asegurándose de que solo las personas autorizadas puedan acceder a ciertas partes de tu sitio.

export function verifyToken(req, res, next) {

    // Obtén el token del encabezado de autorización

    const bearerHeader = req.headers['authorization'];

    if (typeof bearerHeader !== 'undefined') {

        // El token está en el formato "Bearer <token>"

        const token = bearerHeader.split(' ')[1];

        jwt.verify(token, 'secretKey', (error, authData) => {

            if (error) {

                res.status(403).json({ mensaje: 'Token inválido' });

            } else {

                // Si el token es válido, agrega los datos de autenticación a la solicitud para que estén disponibles en las rutas protegidas

                req.authData = authData;

                next();

            }

        });

    } else {

        // Si no se proporcionó un token en el encabezado de autorización

        res.status(403).json({ mensaje: 'Token no proporcionado' });

    }

}

router.get('/products', getProducts);

Esta ruta maneja una solicitud GET a "/products" y llama a la función getProducts para recuperar una lista de productos. No requiere autenticación.

router.post('/products', verifyToken, CreateProducts);

Esta ruta maneja una solicitud POST a "/products" y primero pasa por la función middleware verifyToken antes de llamar a la función CreateProducts. Esto significa que se requiere autenticación para crear un nuevo producto. La función verifyToken verifica si el token JWT proporcionado en el encabezado de autorización es válido antes de permitir que la solicitud se procese más.

//routers

router.get('/products', getProducts);

router.post('/products', verifyToken, CreateProducts);

**Front - End (React)**

**Dependecias**

    "axios": "^1.5.0",

    "env-cmd": "^10.1.0",

    "react": "^18.2.0",

    "react-dom": "^18.2.0",

    "react-router-dom": "^6.16.0",

    "react-scripts": "5.0.1",

    "sweetalert2": "^11.7.31",

    "web-vitals": "^2.1.4"

**axios:** "^1.5.0": Axios es una librería de JavaScript que se utiliza para hacer solicitudes HTTP desde una aplicación frontend a un servidor. Facilita la comunicación entre el frontend y el backend.

**env-cmd:** "^10.1.0": Env-cmd es una herramienta que permite configurar y administrar variables de entorno para una aplicación en diferentes entornos (como desarrollo, producción, pruebas, etc.). Es útil para gestionar configuraciones específicas de cada entorno.

**react:** "^18.2.0": React es una biblioteca de JavaScript ampliamente utilizada para construir interfaces de usuario. Es el núcleo de la mayoría de las aplicaciones de React y proporciona una forma declarativa de construir componentes reutilizables.

**react-dom:** "^18.2.0": React DOM es una biblioteca que se utiliza para interactuar con el DOM (Document Object Model) en aplicaciones web construidas con React. Permite renderizar componentes de React en el navegador.

**react-router-dom:** "^6.16.0": React Router DOM es una extensión de React Router que facilita la navegación y el enrutamiento en aplicaciones React. Permite definir rutas y gestionar la transición entre diferentes vistas de la aplicación.

**react-scripts:** "5.0.1": React Scripts es un conjunto de secuencias de comandos preconfigurados para proyectos de React creados con Create React App. Simplifica tareas como la compilación, el desarrollo local y la creación de proyectos React.

**sweetalert2:** "^11.7.31": SweetAlert2 es una biblioteca que permite crear fácilmente ventanas modales y alertas personalizadas en una aplicación web. Es útil para mostrar mensajes de confirmación, advertencia o éxito de manera atractiva.

**web-vitals**: "^2.1.4": Web Vitals es una biblioteca que ayuda a medir y rastrear el rendimiento web de una aplicación. Proporciona métricas importantes como el tiempo de carga, la interacción del usuario y otros indicadores clave para optimizar el rendimiento de una aplicación web.

Estas dependencias son comunes en proyectos de React y se utilizan para tareas como la gestión de solicitudes HTTP, la navegación en la aplicación, la gestión de variables de entorno, la medición del rendimiento y la creación de interfaces de usuario interactivas. Cada una de ellas desempeña un papel específico en el desarrollo de la aplicación.

**Login**

Este usuario controla la página completamente

*Usuario: Admin*

*Contraseña: Admin1234*

**Registro de empleado**

*Entragador*

*Id: 92*

*Usuario: Entragador92*

*Contraseña: 12345*

*Empleado*

*Id: 25*

*Usuario: Empleado25*

*Contraseña: 12345*

*Admin*

*Id: 90*

*Usuario: Admin*

*Contraseña: Admin12345*

**Encriptar y desencriptar**

Estas dos funciones, encriptar y desencriptar, están diseñadas para cifrar y descifrar datos utilizando el algoritmo de cifrado AES-CBC (Advanced Encryption Standard - Cipher Block Chaining) en JavaScript, utilizando una contraseña proporcionada como clave de cifrado.

Aquí está el desglose de estas dos funciones y las funciones auxiliares utilizadas:

bufferABase64 y base64ABuffer:

bufferABase64 toma un búfer (buffer) y lo convierte en una cadena codificada en base64 utilizando las funciones btoa y String.fromCharCode.

base64ABuffer toma una cadena codificada en base64 y la convierte en un búfer utilizando las funciones atob y Uint8Array.from.

derivacionDeClaveBasadaEnContraseña:

Esta función se utiliza para derivar una clave de cifrado a partir de una contraseña utilizando el algoritmo PBKDF2 (Password-Based Key Derivation Function 2).

Toma una contraseña, una "sal" (salt), un número de iteraciones, una longitud deseada para la clave, un algoritmo hash y un algoritmo de cifrado (por defecto AES-CBC).

Importa la contraseña como una clave bruta y luego utiliza PBKDF2 para derivar la clave utilizando la contraseña, la sal y otros parámetros especificados.

**encriptar**: Esta función se utiliza para cifrar datos utilizando AES-CBC.

Toma una contraseña y un texto plano como entrada.

Genera una "sal" y un vector de inicialización aleatorios utilizando window.crypto.getRandomValues.

Deriva una clave de cifrado utilizando la función derivacionDeClaveBasadaEnContraseña.

Luego, cifra el texto plano utilizando AES-CBC y devuelve el resultado en formato base64, incluyendo la sal y el vector de inicialización en los datos cifrados.

**desencriptar**: Esta función se utiliza para descifrar datos cifrados con AES-CBC.

Toma una contraseña y los datos cifrados en formato base64 como entrada.

Decodifica los datos base64 en un búfer y extrae la sal y el vector de inicialización.

Deriva la clave de cifrado utilizando la función derivacionDeClaveBasadaEnContraseña.

Luego, descifra los datos utilizando AES-CBC y devuelve el texto plano descifrado.

En resumen, estas funciones permiten cifrar y descifrar datos utilizando una contraseña como clave de cifrado, lo que puede ser útil para proteger la confidencialidad de los datos almacenados o transmitidos en una aplicación web.

const bufferABase64 = buffer => btoa(String.fromCharCode(...new Uint8Array(buffer)));

const base64ABuffer = buffer => Uint8Array.from(atob(buffer), c => c.charCodeAt(0));

const LONGITUD\_SAL = 16;

const LONGITUD\_VECTOR\_INICIALIZACION = LONGITUD\_SAL;

// const contraseñaEncriptar = "lfjdnd193016"

// const contraseñaDesencriptar = "lfjdnd193016"

const derivacionDeClaveBasadaEnContraseña  = async (contraseña, sal, iteraciones, longitud, hash, algoritmo = 'AES-CBC') => {

    const encoder = new TextEncoder();

    let keyMaterial = await window.crypto.subtle.importKey(

      'raw',

      encoder.encode(contraseña),

      { name: 'PBKDF2' },

      false,

      ['deriveKey']

    );

    return await window.crypto.subtle.deriveKey(

      {

        name: 'PBKDF2',

        salt: encoder.encode(sal),

        iterations: iteraciones,

        hash

      },

      keyMaterial,

      { name: algoritmo, length: longitud },

      false,

      ['encrypt', 'decrypt']

    );

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_funcion de encriptar\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

export const encriptar = async (contraseña, textoPlano) => {

    const encoder = new TextEncoder();

    const sal = window.crypto.getRandomValues(new Uint8Array(LONGITUD\_SAL));

    const vectorInicializacion = window.crypto.getRandomValues(new Uint8Array(LONGITUD\_VECTOR\_INICIALIZACION));

    const bufferTextoPlano = encoder.encode(textoPlano);

    const clave = await derivacionDeClaveBasadaEnContraseña(contraseña, sal, 100000, 256, 'SHA-256');

    const encrypted = await window.crypto.subtle.encrypt(

      { name: "AES-CBC", iv: vectorInicializacion },

      clave,

      bufferTextoPlano

    );

    return bufferABase64([

      ...sal,

      ...vectorInicializacion,

      ...new Uint8Array(encrypted)

    ]);

};

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_funcion de desencriptar\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

export const desencriptar = async (contraseña, encriptadoEnBase64) => {

    const decoder = new TextDecoder();

    const datosEncriptados = base64ABuffer(encriptadoEnBase64);

    const sal = datosEncriptados.slice(0, LONGITUD\_SAL);

    const vectorInicializacion = datosEncriptados.slice(0 + LONGITUD\_SAL, LONGITUD\_SAL + LONGITUD\_VECTOR\_INICIALIZACION);

    const clave = await derivacionDeClaveBasadaEnContraseña(contraseña, sal, 100000, 256, 'SHA-256');

    const datosDesencriptadosComoBuffer = await window.crypto.subtle.decrypt(

      { name: "AES-CBC", iv: vectorInicializacion },

      clave,

      datosEncriptados.slice(LONGITUD\_SAL + LONGITUD\_VECTOR\_INICIALIZACION)

    );

    return decoder.decode(datosDesencriptadosComoBuffer);

  }

**<BrowserRouter>:** Esto es un componente proporcionado por React Router que se utiliza para envolver toda la estructura de enrutamiento de la aplicación. <BrowserRouter> habilita el enrutamiento basado en el navegador, lo que significa que la URL en la barra de direcciones del navegador se utilizará para determinar qué componente se debe mostrar en la aplicación.

**<Routes>:** Dentro del <BrowserRouter>, tienes un componente <Routes>. Este componente es responsable de definir todas las rutas posibles de tu aplicación y mapear esas rutas a los componentes que se deben renderizar cuando se visita una ruta específica.

**<Route>:** Cada <Route> es una definición de una ruta específica de tu aplicación. Tienes varias rutas definidas en tu aplicación, cada una de las cuales tiene un path (ruta) y un element (elemento a renderizar cuando se accede a esa ruta). Aquí hay una descripción de cada una de las rutas:

*/: La ruta raíz de tu aplicación. Cuando la URL coincide con "/", se renderiza el componente <Home>.*

*/productos: Cuando la URL coincide con "/productos", se renderiza el componente <Products>.*

*/login: Cuando la URL coincide con "/login", se renderiza el componente <Login>.*

*/carrito: Cuando la URL coincide con "/carrito", se renderiza el componente <Car>.*

*/pedidos: Cuando la URL coincide con "/pedidos", se renderiza el componente <Order>.*

*/datos: Cuando la URL coincide con "/datos", se renderiza el componente <Data>.*

*/empleados: Cuando la URL coincide con "/empleados", se renderiza el componente <Workers>.*

*/clientes: Cuando la URL coincide con "/clientes", se renderiza el componente <Client>.*

*/ventas: Cuando la URL coincide con "/ventas", se renderiza el componente <Sales>.*

Ruta de error **(<Route path="/\*" element={<Err />} />):** Esta ruta es un comodín y captura cualquier URL que no coincida con ninguna de las rutas anteriores. Cuando la URL no coincide con ninguna de las rutas definidas, se renderiza el componente <Err>, que destinado a mostrar una página de error 404.

En resumen, este fragmento de código establece las rutas y los componentes correspondientes que deben renderizarse cuando se accede a diferentes URL en tu aplicación web. Es una parte esencial de la configuración de React Router para habilitar la navegación entre páginas o vistas en tu aplicación web.

   <BrowserRouter>

          <Routes>

            {/\*content\*/}

            <Route path="/" element={<Home />} />

            <Route path="/productos" element={<Products />} />

            <Route path="/login" element={<Login />} />

            <Route path="/carrito" element={<Car />} />

            <Route path="/pedidos" element={<Order />} />

            <Route path="/datos" element={<Data />} />

            <Route path="/empleados" element={<Workers />} />

            <Route path="/clientes" element={<Client />} />

            <Route path="/ventas" element={<Sales />} />

            {/\* Protect routers Err 404\*/}

            <Route path="/\*" element={<Err />} />

          </Routes>

        </BrowserRouter>

**JWT**

**axios.post():** Esta línea utiliza Axios para realizar una solicitud HTTP POST a la URL especificada. Axios es una librería de JavaScript ampliamente utilizada para hacer solicitudes HTTP desde aplicaciones frontend a servidores.

**'http://localhost:3030/api/products'**: Esta es la URL a la que se enviará la solicitud POST. En este caso, la solicitud se está enviando **a http://localhost:3030/api/products,** lo que implica que la aplicación está interactuando con una API en el servidor local en el puerto 3030 y tratando de crear o actualizar un recurso relacionado con productos.

**data:** data es el segundo argumento de la función **axios.post().** Es el cuerpo de la solicitud POST y generalmente contiene los datos que se enviarán al servidor para crear o actualizar un recurso. La estructura y contenido de data dependerán de los requisitos específicos de la API a la que estás haciendo la solicitud.

**headers:** Este es el tercer argumento de la función **axios.post()** y es un objeto que define las cabeceras HTTP que se incluirán en la solicitud. En este caso, se está configurando un encabezado de autorización (Authorization) que generalmente se usa para enviar un token de autenticación al servidor.

**"Authorization": Bearer ${localStorage.getItem("token")}``:** Esta línea configura la cabecera de autorización de la solicitud. En ella, se utiliza el valor del token almacenado en el almacenamiento local **(localStorage.getItem("token"))** y se coloca dentro del encabezado utilizando el formato **"Bearer".** Este tipo de encabezado de autorización se utiliza comúnmente para autenticar las solicitudes en una API. El token debe haber sido obtenido previamente durante un proceso de autenticación en el frontend y almacenado en el almacenamiento local para su posterior uso.

        axios.post('http://localhost:3030/api/products', data, {

            headers: {

                "Authorization": `Bearer ${localStorage.getItem("token")}`

            }

        })

**Vistas**

**Productos**

Generación de un código aleatorio: En la función See, se genera un código aleatorio y se almacena en el estado randomN. Este código parece estar relacionado con la creación de una orden o producto.

Gestión del carrito de compras: Existe una función llamada add que permite agregar productos al carrito de compras. Los productos se almacenan tanto en el estado local como en el estado del componente (products\_2). También se muestra una notificación de éxito utilizando SweetAlert2 cuando se agrega un producto al carrito.

Gestión de marcas y categorías: Se tienen funciones como SeeBrand y SeeCategory que muestran componentes relacionados con la creación de marcas y categorías. Además, hay funciones (createBrand, createCategory) para crear marcas y categorías en la base de datos

Gestión de la edición de productos: La función SeeEdit muestra un componente relacionado con la edición de productos y parece almacenar datos en el almacenamiento local.

Limpieza de los campos de entrada: La función Ocult oculta algunos componentes y también limpia los campos de entrada mediante llamadas a funciones set para establecer los estados en valores vacíos.

Estado y funciones para recopilar datos del formulario: Se definen varios estados (por ejemplo, Nproduct, Dproduct, Bproduct, etc.) para recopilar datos del formulario. Cada campo de entrada tiene su propia función onChange que actualiza el estado correspondiente cuando se cambian los valores en los campos de entrada.

Filtrado de productos: Se realiza un filtrado de productos basado en el texto de búsqueda, la marca seleccionada y la categoría seleccionada. Los productos filtrados se almacenan en el estado filteredCharacters. El resultado del filtrado se actualiza cada vez que cambian los valores de búsqueda, marca o categoría.

Creación, actualización y eliminación de productos: Se tienen funciones para crear (createProduct), actualizar (update) y eliminar (deleteProduct) productos en la base de datos. Se utilizan solicitudes HTTP para realizar estas operaciones.

****

**Pedidos**

Estados Iniciales: Se definen varios estados iniciales utilizando useState para almacenar datos relacionados con los pedidos y la asignación de entregadores.

Obtener Todos los Pedidos: Se utiliza un efecto (useEffect) para hacer una solicitud GET a una API (http://localhost:3030/api/order) y obtener todos los pedidos almacenados en la base de datos. Los pedidos se almacenan en el estado order. Se verifica si hay un token en el almacenamiento local antes de realizar la solicitud para asegurarse de que el usuario esté autenticado

Filtrar Pedidos: Hay una función llamada filterOrders que filtra los pedidos en función del texto de búsqueda (text) y las fechas de inicio y fin seleccionadas (selectedStartDate y selectedEndDate). Los resultados del filtrado se almacenan en el estado filteredOrders.

Cancelar Pedido: La función deleteOrders permite cancelar un pedido mediante una solicitud DELETE a la API (http://localhost:3030/api/order/{idOrder}). Se utiliza el token de autorización para autenticar la solicitud. Si se completa con éxito, se recarga la página después de 1 segundo y se muestra un mensaje de éxito.

Registrar Venta: La función sales parece estar relacionada con el registro de ventas. Se obtienen detalles de un pedido en función del ID del pedido (idOrder) y se construye un objeto de datos con ciertos atributos. Luego, se realiza una solicitud POST a la API (http://localhost:3030/api/sales) para registrar la venta. Si se completa con éxito, se muestra un mensaje de éxito.

Asignar Entregador: Se utiliza un conjunto de estados (asig, asigId) para asignar un entregador a un pedido. La función seeAsing muestra un componente relacionado con la asignación. La función Asig realiza una solicitud PUT a la API para actualizar los detalles del pedido con el ID especificado (asigId) e incluir el ID del entregador asignado (asig). También se recarga la página después de realizar la asignación.

****

**Empleado**

Obtener Todos los Trabajadores: Se utiliza el estado work y un efecto (useEffect) para hacer una solicitud GET a una API (http://localhost:3030/api/work/workers) y obtener todos los trabajadores almacenados en la base de datos. Los trabajadores se almacenan en el estado work. Se verifica si hay un token en el almacenamiento local antes de realizar la solicitud para asegurarse de que el usuario esté autenticado.

Obtener Cantidad de Trabajadores: Similar al punto anterior, se utiliza el estado count y otro efecto para obtener la cantidad total de trabajadores. La solicitud se hace a http://localhost:3030/api/work/workers/count.

Eliminar Trabajador: La función deleteWorker permite eliminar un trabajador mediante una solicitud DELETE a la API (http://localhost:3030/api/work/workers/{idWork}). Se utiliza el token de autorización para autenticar la solicitud. Si se completa con éxito, se recarga la página después de 1 segundo y se muestra un mensaje de éxito.

Filtrar Trabajadores: Se utiliza un campo de texto (text) para filtrar los trabajadores en función del nombre. La función inputLoad se utiliza para actualizar el estado text cuando el usuario escribe en el campo de búsqueda. Los trabajadores que coinciden con el filtro se almacenan en inputCharacters.

Estados y Cambios en los Datos del Trabajador: Se definen múltiples estados (Nwork, Awork, CEwork, etc.) para capturar los datos del nuevo trabajador que se ingresará en un formulario. Cada onChange está vinculado a un estado específico para actualizarlo cuando el usuario ingrese información en el formulario.

Mostrar y Ocultar Formulario: Las funciones See y Ocult se utilizan para mostrar y ocultar un formulario de ingreso de datos de trabajadores cuando sea necesario. Al ocultar el formulario, se limpian los campos de entrada (setNwork, setAwork, etc.).

Enviar Datos del Trabajador: La función sendData se utiliza para enviar los datos del nuevo trabajador al servidor mediante una solicitud POST a http://localhost:3030/api/work/workers. Se utilizan los estados Nwork, Awork, etc. para recopilar los datos. El token de autorización se incluye en el encabezado de la solicitud. Se muestra la respuesta de la API en la consola.

****

**Carrito**

Obtención de datos del carrito desde el almacenamiento local: Se utiliza localStorage.getItem("car") para obtener los datos del carrito de compras almacenados en el almacenamiento local. Si hay datos en el carrito (carData no es nulo), se parsean como objetos JSON y se almacenan en la variable productsInCart. Si no hay datos en el carrito, se inicializa productsInCart como un arreglo vacío.

Función deleteCar: Esta función se ejecuta cuando se intenta eliminar un producto del carrito. Se utiliza la librería SweetAlert2 para mostrar una confirmación al usuario. Si el usuario confirma la eliminación, se realiza lo siguiente:

Se busca el índice del producto a eliminar en el arreglo productsInCart utilizando findIndex.

Si se encuentra el producto en el carrito (índice distinto de -1), se elimina ese producto del arreglo utilizando splice.

El arreglo actualizado se guarda nuevamente en el almacenamiento local como datos del carrito.

Se utiliza window.location.reload() para recargar la página, reflejando los cambios en el carrito.

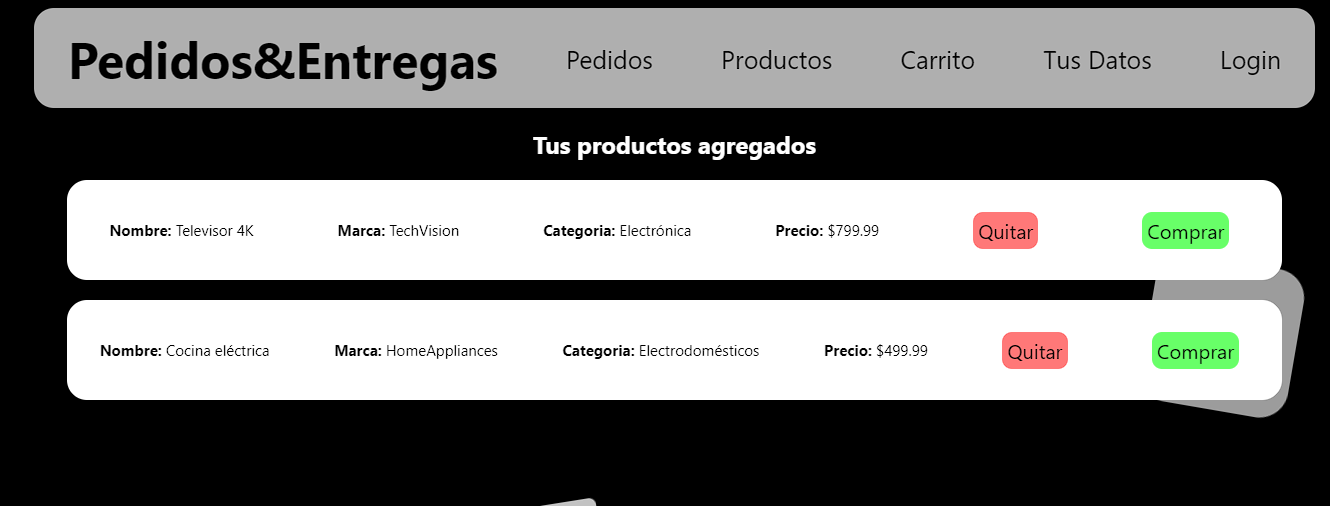
Función add\_2: Esta función se utiliza para agregar productos al carrito. Al hacer clic en un botón o elemento en la interfaz de usuario, se llama a esta función y se pasa un valor e.target.value. Esta función realiza lo siguiente:

Verifica si ya hay datos en el almacenamiento local para el producto actual. Si no hay datos, inicializa validDatos\_2 como un arreglo vacío.

Almacena los datos del producto actual (valor de e.target.value) en validDatos\_2.

Guarda validDatos\_2 en el almacenamiento local bajo la clave "product".

En resumen, este código se encarga de la gestión de un carrito de compras, permitiendo a los usuarios agregar y eliminar productos del carrito. Los datos del carrito se almacenan en el almacenamiento local del navegador para persistencia.

****

**Ventas**

Estado para Almacenar Ventas: Se utiliza el estado ventas para almacenar los datos de ventas obtenidos de una API. Inicialmente, se establece como un arreglo vacío.

Filtrar Ventas por Texto: Se utiliza el estado text para capturar el texto ingresado por el usuario en un campo de búsqueda. La función inputLoad se llama cada vez que el usuario ingresa texto en el campo y actualiza el estado text en consecuencia.

Obtener Datos de Ventas: Se utiliza un efecto (useEffect) para hacer una solicitud GET a la API cuya URL es http://localhost:3030/api/sales. Se incluye un encabezado de autorización con un token obtenido del almacenamiento local. Los datos de ventas se almacenan en el estado ventas.

Funciones para Calcular el Total de Ventas: Se definen tres funciones (calcularTotalPorDia, calcularTotalPorSemana, calcularTotalPorMes) que se utilizan para calcular el total de ventas para un día, una semana y un mes específicos, respectivamente. Estas funciones toman como entrada el arreglo de ventas y una fecha.

calcularTotalPorDia: Filtra las ventas que coinciden con la fecha dada y suma sus valores para obtener el total de ventas de ese día.

calcularTotalPorSemana: Filtra las ventas dentro de la semana que contiene la fecha dada y calcula el total de ventas.

calcularTotalPorMes: Filtra las ventas dentro del mes que contiene la fecha dada y calcula el total de ventas.

Calcular Totales y Mostrar Resultados: Se utilizan las funciones de cálculo de ventas para calcular los totales de ventas por día, semana y mes, utilizando la fecha almacenada en text. Los resultados se almacenan en las variables totalPorDia, totalPorSemana y totalPorMes. Estos valores se muestran en la consola con console.log.

Obtener el ID del Trabajador: Se utiliza localStorage para obtener el ID del trabajador, que parece ser relevante para la aplicación.

****

**Clientes**

Estado para Almacenar Clientes: Se utiliza el estado clients para almacenar los datos de clientes obtenidos de una API. Inicialmente, se establece como un arreglo vacío.

Estado para Clientes Filtrados: El estado filteredClients se utiliza para almacenar los datos de clientes después de aplicar filtros de búsqueda. También se inicializa como un arreglo vacío

Estados para Búsqueda y Filtros: Se utilizan varios estados (text, selectedDepartamento, selectedMunicipio, activoFilter) para capturar los valores de búsqueda y los filtros aplicados por el usuario. La función inputLoad se llama cada vez que el usuario ingresa texto en un campo de búsqueda para actualizar el estado text.

Obtener Datos de Clientes: Se utiliza un efecto (useEffect) para hacer una solicitud GET a la API cuya URL es http://localhost:3030/api/client. Se incluye un encabezado de autorización con un token obtenido del almacenamiento local. Los datos de clientes se almacenan en el estado clients.

Obtener Datos de Departamentos y Municipios: Se utilizan efectos adicionales para obtener datos de departamentos y municipios de la API. Los datos se almacenan en los estados Depar y Muni, respectivamente.

Función para Filtrar Clientes: La función filterClients se utiliza para aplicar filtros a los datos de clientes según los valores de búsqueda y filtros seleccionados. Los clientes que coincidan con los criterios de búsqueda se almacenan en el estado filteredClients.

Efecto para Filtrar Clientes: Se utiliza un efecto adicional para llamar a la función filterClients cada vez que cambian los estados relacionados con la búsqueda y los filtros (clients, text, selectedDepartamento, selectedMunicipio, activoFilter).

Funciones para Inactivar y Activar Clientes: Las funciones Inactive y Active se utilizan para cambiar el estado de activo/inactivo de un cliente. Estas funciones hacen solicitudes PUT a la API para actualizar el estado de un cliente a activo o inactivo según el valor proporcionado.

Obtener el ID del Trabajador: Se utiliza localStorage para obtener el ID del trabajador, que parece ser relevante para la aplicación.

****

**Iniciar sesión**

Generación de un número de orden aleatorio al montar el componente: En este fragmento de código, se utiliza el useState y el useEffect de React para generar un número de orden aleatorio cuando el componente se monta. El número de orden se almacena en el estado randomN. El uso del arreglo vacío [] en el useEffect asegura que esta generación de número de orden ocurra solo una vez cuando el componente se monta.

Definición de estados para campos de formulario: Se definen estados para varios campos del formulario, como departamento, ciudad, barrio, direccionEntrega, y Obser. Cada uno de estos estados se inicializa como cadenas vacías.

Obtención de datos de productos desde el almacenamiento local: Se obtienen datos relacionados con productos desde el almacenamiento local (localStorage). Estos datos incluyen carData, Product, y productsInCart. carData se utiliza para obtener información del carrito de compras almacenada en el localStorage. Product parece ser un identificador de producto. productsInCart se utiliza para almacenar la información de los productos en el carrito de compras.

Actualización de estados basada en los datos del producto: Se itera a través de productsInCart para encontrar el producto correspondiente al identificador Product. Luego, se actualizan los estados relacionados con ese producto, como nombreProducto, codigoProducto, marca, categoria, iva, y Price.

Función createOrder: Esta función se utiliza para crear una orden de compra. Se crea un objeto Data que contiene la información de la orden, incluyendo el número de orden generado aleatoriamente (randomN), el precio total (Price), el IVA (iva), etc. Luego, se realiza una solicitud POST a una URL específica para crear la orden en el servidor.

Función details: Esta función se utiliza para crear detalles de la orden, como el nombre del producto, la descripción, el IVA, el precio, etc. Luego, realiza una solicitud POST para crear estos detalles en el servidor. Después de crear los detalles, se llama a la función createOrder para crear la orden principal. También hay un redireccionamiento después de completar la acción.

Funciones Next y Ocult: Estas funciones se utilizan para mostrar y ocultar un elemento en la interfaz de usuario identificado por su ID (postDataMain). Esto parece estar relacionado con la interfaz de usuario de la aplicación.

En resumen, permite a los usuarios realizar órdenes de compra y almacenar detalles de productos relacionados con esas órdenes. También muestra y oculta elementos de la interfaz de usuario según sea necesario.

****