CARD CUSTOMS



***https://github.com/jCanay/card-customs***

**Nome Alumno/a: Jaime Canay Agho**

***Jaime Canay Agho***

**Curso: *1º DAM* Materia:** ***Bases de Datos – Proyecto Final 24/25***

Contido

[Introducción 2](#_Toc199615594)

[1. Descripción del Problema / Requisitos 2](#_Toc199615595)

[2. Modelo Conceptual 3](#_Toc199615596)

[3. Modelo Relacional 4](#_Toc199615597)

[4. Proceso de Normalización 4](#_Toc199615598)

[5. Script de Creación de la Base de Datos 6](#_Toc199615599)

[6. Carga de Datos Inicial 9](#_Toc199615600)

[7. Funciones y Procedimientos Almacenados 9](#_Toc199615601)

[8. Triggers 13](#_Toc199615602)

[9. Consultas SQL 16](#_Toc199615603)

[10. Casos de Prueba y Simulación 18](#_Toc199615604)

[11. Resultados y Verificación 18](#_Toc199615605)

[12. Capturas de Pantalla (opcional) 18](#_Toc199615606)

[13. Conclusiones y Mejoras Futuras 18](#_Toc199615607)

[14. Enlace al Repositorio en GitHub 18](#_Toc199615608)

# Introducción

El presente proyecto se centra en el diseño e implementación de un **sistema de** **gestión de bases de datos relacional** para un negocio dedicado a la personalización y venta de cartas. Con el objetivo de optimizar y centralizar las operaciones diarias, este sistema busca ofrecer una plataforma robusta para la administración de productos, pedidos, clientes, empleados y el control de inventario.

Desarrollado usando MySQL, la base de datos card\_customs ha sido diseñada para garantizar la integridad, consistencia y disponibilidad de la información, permitiendo una gestión eficiente de los flujos de trabajo clave del negocio. Este documento detalla los requisitos, el proceso de creación y la funcionalidad de la base de datos, así como las decisiones técnicas adoptadas para cumplir con los requisitos del proyecto.

# Descripción del Problema / Requisitos

Se necesita crear una base de datos para poder gestionar una tienda online que vende diferentes productos relacionados con juegos de mesa.

Card Customs es una tienda especializada en personalización artística de cartas de juegos de mesa, principalmente Magic: The Gathering. Necesitan una base de datos para gestionar:

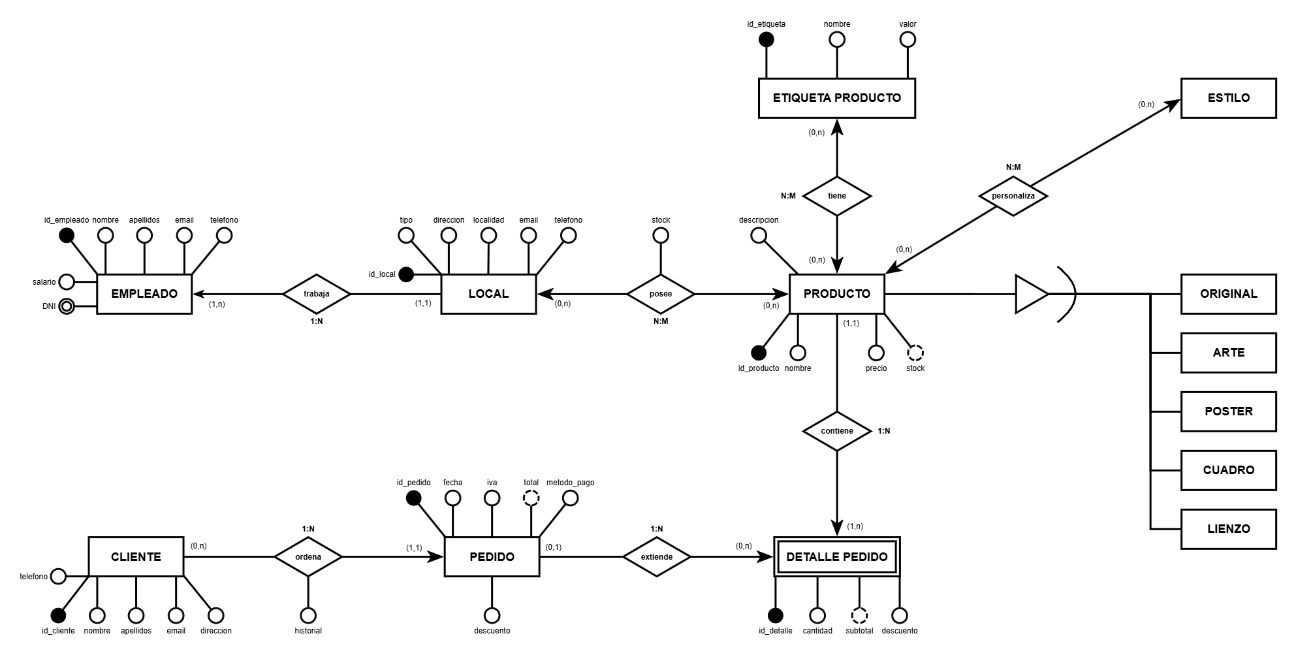
* Inventario de cartas disponibles
* Catálogo de diseños predefinidos
* Pedidos de clientes
* Artistas y sus trabajos
* Ubicaciones (actualmente online y un taller físico)
* Servicios ofrecidos

La base de datos debe estar preparada para una posible expansión futura con más ubicaciones físicas. El sistema debe permitir rastrear todo el proceso desde que un cliente hace un pedido hasta que se completa el trabajo de personalización, incluyendo qué artista lo realizó, qué diseño se usó, y dónde se realizó el trabajo.

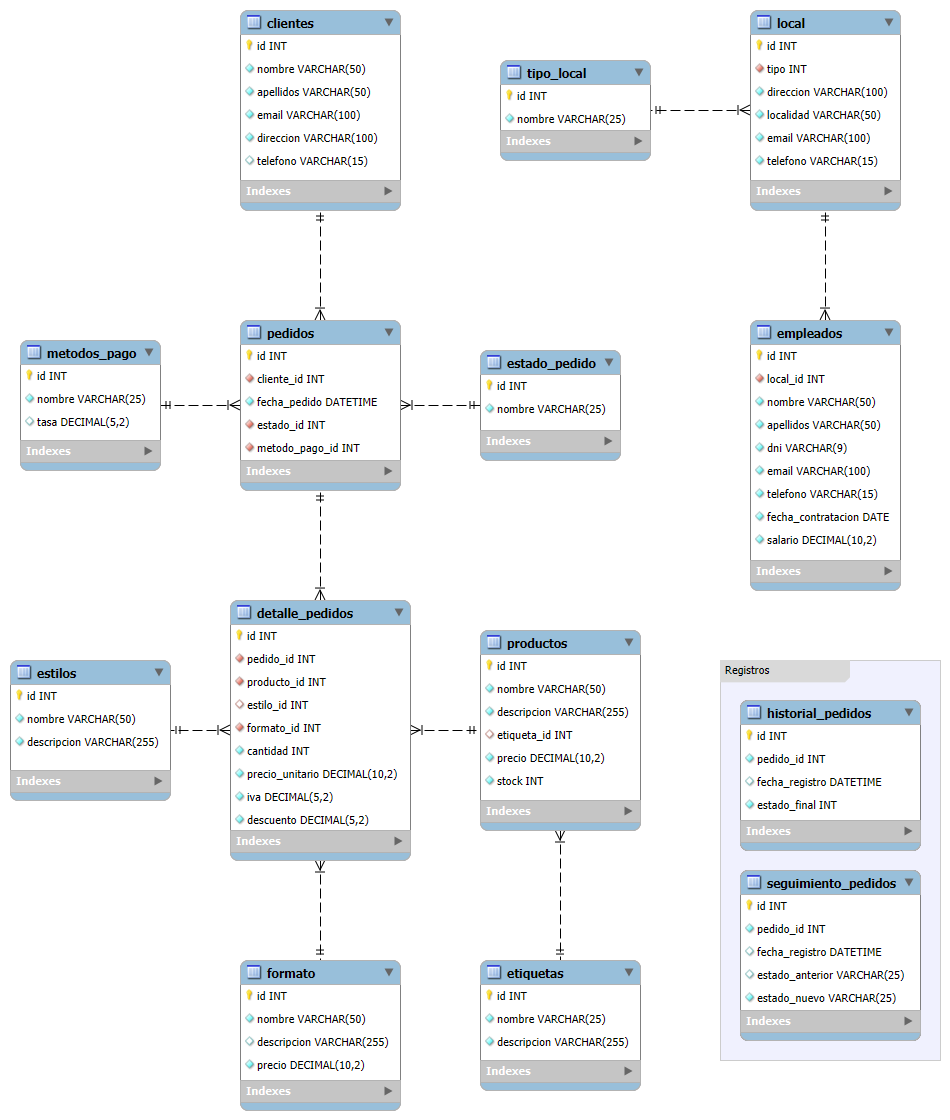
El sistema debe cumplir los siguientes requisitos para su correcto funcionamiento:

|  |  |
| --- | --- |
| **REQUISITOS FUNCIONALES**  1. Gestión de Clientes   * Registrar nuevos clientes * Actualizar información de clientes * Consultar historial de pedidos por cliente   2. Gestión de Pedidos   * Crear nuevos pedidos * Modificar estado de pedidos * Registrar detalles de personalización * Calcular precios totales * Seguimiento del estado del pedido   3. Gestión de Productos   * Mantener inventario * Gestionar catálogo de productos * Control de stock * Asignar etiquetas a productos   4. Gestión de Personal   * Registrar empleados * Asignar empleados a locales * Gestionar información laboral   5. Gestión de Locales   * Administrar diferentes tipos de locales * Mantener información de contacto * Gestionar ubicaciones | **REQUISITOS NO FUNCIONALES**  1. Rendimiento   * + Tiempo de respuesta rápido en consultas   + Capacidad para manejar múltiples transacciones simultáneas   2. Seguridad   * + Protección de datos sensibles (DNI, salarios)   + Control de acceso por roles   + Registro de cambios en pedidos   3. Disponibilidad   * Sistema operativo 24/7 * Backup regular de datos   4. Escalabilidad   * Capacidad para crecer con el negocio * Soporte para múltiples locales   5. Mantenibilidad   * Estructura normalizada * Documentación clara * Facilidad de actualización |

# Modelo Conceptual



# Modelo Relacional



# Proceso de Normalización

El diseño de la base de datos Card Customs se ha creado siguiendo los principios de la normalización, llegando específicamente hasta la Tercera Forma Normal (3FN). Esta forma de diseñar se eligió para mejorar la gestión de los datos, buscando que sean eficientes, correctos y fáciles de usar.

**¿Qué Implica la Tercera Forma Normal (3FN)?**

La 3FN es una manera de organizar bases de datos que asegura que una tabla, además de cumplir con las dos primeras formas normales (1FN y 2FN), no tenga dependencias indirectas entre datos. Esto quiere decir que una columna que no es la clave principal no debe depender de otra columna que tampoco sea la clave principal. En resumen, cada dato en una tabla debe depender solo de la clave principal de esa tabla.

**Beneficios Aplicados al Diseño de Card Customs:**

Usar la 3FN en nuestra base de datos trae muchas ventajas para cómo funciona y cómo está hecha:

* **Reducción de Datos Repetidos:** Al organizar la información en partes separadas, se reduce mucho que los datos se repitan. Por ejemplo, en lugar de guardar varias veces la información completa de un cliente (nombre, dirección, teléfono, email) en cada registro de la tabla pedidos, la tabla clientes guarda estos datos una sola vez. La tabla pedidos simplemente usa el número de identificación del cliente que ya existe (*cliente\_id*). Este diseño reduce bastante el espacio que se necesita para almacenar datos y hace más fácil manejar la información.
* **Evitar Problemas en las Operaciones de Datos:** La 3FN es muy importante para que no haya errores cuando se añaden, cambian o borran datos:
  + **Problemas al Añadir Datos Nuevos:** Permite guardar datos de algo (por ejemplo, un nuevo *tipo\_local*, un *estilo* o un *formato*) incluso si aún no ha sido usado en una tabla más importante (como un local o un *detalle\_pedidos*). Tener tablas especiales para estas entidades (*tipo\_local*, *estado\_pedido*, *metodos\_pago*, *estilos*, *formato*) hace que podamos definir estos elementos sin que ya estén usándose en otras tablas.
  + **Problemas al Cambiar Datos:** Si un dato está en muchos sitios sin una buena organización, cualquier cambio en ese dato haría que se tuviera que cambiar en todos los sitios donde aparece. Por ejemplo, si los datos de un proveedor estuvieran copiados en la tabla productos, un cambio en la dirección del proveedor significaría que tendríamos que cambiar cada producto que ese proveedor vende. Con la 3FN y la existencia de una tabla proveedores, la actualización se hace en un solo sitio, asegurando que el dato sea siempre el mismo en toda la base de datos.
  + **Problemas al Borrar Datos:** Evita que se borre información sin querer. Si la información de un cliente solo existiera en la tabla pedidos (y no en una tabla clientes separada), borrar el último pedido de ese cliente haría que se perdiera toda su información. La 3FN asegura que borrar un dato de una operación (como un pedido) no borra el dato principal al que se refiere (como el cliente).
* **Mejora de la Fiabilidad y Coherencia de los Datos:** El uso de claves foráneas, que conectan las tablas entre sí (por ejemplo, *pedido\_id* en *detalle\_pedidos* que apunta a *pedidos*), hace que los enlaces entre los datos sean correctos. Esto garantiza que los datos se conecten bien entre sí y que, por ejemplo, un *detalle\_pedido* siempre estará ligado a un producto, estilo y formato que de verdad existen. Esto ayuda a que la información sea siempre de confianza en la base de datos.
* **Mayor Claridad y Facilidad de Mantenimiento:** Una base de datos bien organizada está mejor hecha y es más fácil de entender. Cada tabla tiene un objetivo claro y sus datos están juntos de forma lógica. Esta organización hace más fácil que la gente que trabaja con la base de datos la entienda, y también simplifica añadir cosas nuevas o arreglar fallos.

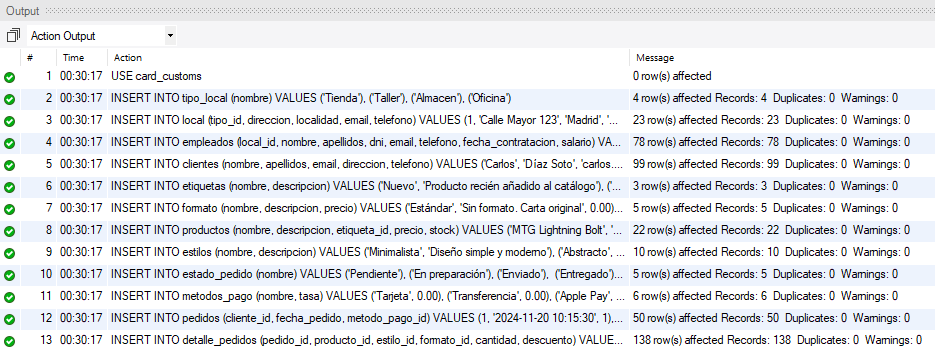
En resumen, seguir la Tercera Forma Normal en el diseño de la base de datos Card Customs es muy importante para crear un sistema de información fuerte, que pueda crecer y con datos de muy buena calidad.

# Script de Creación de la Base de Datos

|  |
| --- |
| CREATE DATABASE IF NOT EXISTS card\_customs DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  USE card\_customs;  -- Creación de tablas  CREATE TABLE IF NOT EXISTS tipo\_local (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(25) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS local (      id INT AUTO\_INCREMENT PRIMARY KEY,      tipo\_id INT NOT NULL,      direccion VARCHAR(100) NOT NULL,      localidad VARCHAR(50) NOT NULL,      email VARCHAR(100) NOT NULL UNIQUE,      telefono VARCHAR(15) NOT NULL,      FOREIGN KEY (tipo\_id) REFERENCES tipo\_local(id) ON DELETE RESTRICT ON UPDATE CASCADE  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS empleados (      id INT AUTO\_INCREMENT PRIMARY KEY,      local\_id INT NOT NULL,      nombre VARCHAR(50) NOT NULL,      apellidos VARCHAR(50) NOT NULL,      dni VARCHAR(9) NOT NULL UNIQUE,      email VARCHAR(100) NOT NULL UNIQUE,      telefono VARCHAR(15) NOT NULL,      fecha\_contratacion DATE NOT NULL,      salario DECIMAL(10, 2) NOT NULL,      FOREIGN KEY (local\_id) REFERENCES local(id) ON DELETE RESTRICT ON UPDATE CASCADE  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS clientes (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(50) NOT NULL,      apellidos VARCHAR(50) NOT NULL,      email VARCHAR(100) NOT NULL UNIQUE,      direccion VARCHAR(100) NOT NULL,      telefono VARCHAR(15)  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS etiquetas (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(25) NOT NULL,      descripcion VARCHAR(255) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS formato (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(50) NOT NULL,      descripcion VARCHAR(255),      precio DECIMAL(10, 2) CHECK (precio >= 0) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS productos (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(50) NOT NULL,      descripcion VARCHAR(255) NOT NULL,      etiqueta\_id INT,      precio DECIMAL(10, 2) NOT NULL,      stock INT NOT NULL CHECK (stock >= 0),      FOREIGN KEY (etiqueta\_id) REFERENCES etiquetas(id) ON DELETE SET NULL ON UPDATE CASCADE  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS estilos (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(50) NOT NULL,      descripcion VARCHAR(255) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS estado\_pedido (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(25) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS metodos\_pago (      id INT AUTO\_INCREMENT PRIMARY KEY,      nombre VARCHAR(25) NOT NULL,      tasa DECIMAL(5, 2)  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS pedidos (      id INT AUTO\_INCREMENT PRIMARY KEY,      cliente\_id INT NOT NULL,      fecha\_pedido DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,      estado\_id INT NOT NULL DEFAULT 1,      metodo\_pago\_id INT NOT NULL,      FOREIGN KEY (cliente\_id) REFERENCES clientes(id) ON DELETE CASCADE ON UPDATE CASCADE,      FOREIGN KEY (estado\_id) REFERENCES estado\_pedido(id) ON DELETE RESTRICT ON UPDATE CASCADE,      FOREIGN KEY (metodo\_pago\_id) REFERENCES metodos\_pago(id) ON DELETE RESTRICT ON UPDATE RESTRICT  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS detalle\_pedidos (      id INT AUTO\_INCREMENT PRIMARY KEY,      pedido\_id INT NOT NULL,      producto\_id INT NOT NULL,      estilo\_id INT,      formato\_id INT NOT NULL,      cantidad INT NOT NULL CHECK (cantidad > 0),      precio\_unitario DECIMAL(10, 2) NOT NULL CHECK precio\_unitario > 0,      iva DECIMAL(5, 2) NOT NULL DEFAULT 21.00 CHECK iva > 0,      descuento DECIMAL(5, 2) NOT NULL DEFAULT 0,      FOREIGN KEY (pedido\_id) REFERENCES pedidos(id) ON DELETE RESTRICT ON UPDATE CASCADE,      FOREIGN KEY (producto\_id) REFERENCES productos(id) ON DELETE RESTRICT ON UPDATE CASCADE,      FOREIGN KEY (estilo\_id) REFERENCES estilos(id) ON DELETE SET NULL ON UPDATE CASCADE,      FOREIGN KEY (formato\_id) REFERENCES formato(id) ON DELETE RESTRICT ON UPDATE CASCADE  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS historial\_pedidos (      id INT AUTO\_INCREMENT PRIMARY KEY,      pedido\_id INT NOT NULL,      fecha\_registro DATETIME,      estado\_final INT NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci;  CREATE TABLE IF NOT EXISTS seguimiento\_pedidos (      id INT AUTO\_INCREMENT PRIMARY KEY,      pedido\_id INT NOT NULL,      fecha\_registro DATETIME,      estado\_anterior VARCHAR(25),      estado\_nuevo VARCHAR(25) NOT NULL  ) ENGINE InnoDB DEFAULT CHARSET utf8mb4 COLLATE utf8mb4\_0900\_ai\_ci; |

# Carga de Datos Inicial

Se han insertado múltiples datos ficticios para realizar las pruebas de la base de datos y comprobar su correcto funcionamiento ante cualquier caso de uso



# Funciones y Procedimientos Almacenados

***FUNCIONES***

|  |
| --- |
| USE card\_customs;  -- Esta función se encarga de calcular el subtotal de cada factura teniendo en cuenta un 'precio', un 'formato\_precio' una 'cantidad', un 'iva' y un 'descuento'.  DELIMITER //  CREATE FUNCTION calcular\_subtotal\_factura (precio DECIMAL(10,2), formato\_precio DECIMAL(10,2), cantidad INT, iva DECIMAL(5,2), descuento DECIMAL(5,2))  RETURNS DECIMAL(10,2)  DETERMINISTIC  BEGIN      DECLARE precio\_formato DECIMAL (10,2) DEFAULT (precio + formato\_precio);      DECLARE precio\_iva DECIMAL(10,2) DEFAULT (precio\_formato + (precio\_formato \* iva / 100));      DECLARE precio\_iva\_descuento DECIMAL(10, 2) DEFAULT (precio\_iva - (precio\_iva \* descuento / 100));      DECLARE subtotal DECIMAL(10,2) DEFAULT (precio\_iva\_descuento \* cantidad);        RETURN subtotal;  END;  // DELIMITER ;  -- Esta función se encarga de calcular el total de cada factura. Normalmente al argumento 'subtotal' se le pasará el resultado de la función 'calcular\_subtotal\_factura'.  DELIMITER //  CREATE FUNCTION calcular\_total\_factura (subtotal DECIMAL(10,2), tasa DECIMAL(5,2))  RETURNS DECIMAL(10,2)  DETERMINISTIC  BEGIN      DECLARE total DECIMAL(10,2) DEFAULT (subtotal + (subtotal \* tasa / 100));      RETURN total;  END;  // DELIMITER ;  -- Esta función se encarga de obtener el stock de un producto pasado por parámetro.  DELIMITER //  CREATE FUNCTION obtener\_stock\_producto(producto\_id INT)  RETURNS INT  DETERMINISTIC  BEGIN      RETURN (SELECT stock FROM productos WHERE id = producto\_id);  END;  // DELIMITER ;  -- Esta función se encarga de obtener el nombre de un estado, que se pasa como parámetro, de la tabla 'estado\_pedido'.  DELIMITER //  CREATE FUNCTION obtener\_nombre\_estado(estado\_id INT)  RETURNS VARCHAR(25)  DETERMINISTIC  BEGIN      RETURN (SELECT nombre FROM estado\_pedido WHERE id = estado\_id);  END;  // DELIMITER ;  -- Esta función devuelve el salario del empleado con mayor salario.  DELIMITER //  CREATE FUNCTION mayor\_salario\_empleados()  RETURNS DECIMAL(10,2)  DETERMINISTIC  BEGIN      RETURN (SELECT e.salario FROM empleados e ORDER BY salario DESC LIMIT 1);  END;  // DELIMITER ; |

***PROCEDIMIENTOS***

|  |
| --- |
| USE card\_customs;  -- Este procedimiento se encargar de mostrar todos los datos de una factura especificada y mostrarlos de una forma que se entienda mejor cada fila.  DELIMITER //  CREATE PROCEDURE obtener\_factura\_detallada\_pedido(pedido\_id INT)  BEGIN      SELECT pe.id, pe.fecha\_pedido, c.nombre, c.apellidos, pr.nombre, f.nombre, e.nombre, dp.precio\_unitario, dp.cantidad, dp.iva, dp.descuento      FROM detalle\_pedidos dp JOIN pedidos pe ON pe.id = dp.pedido\_id JOIN clientes c ON c.id = pe.cliente\_id JOIN formato f ON f.id = dp.formato\_id LEFT JOIN estilos e ON dp.estilo\_id = e.id JOIN productos pr ON pr.id = dp.producto\_id      WHERE dp.pedido\_id = pedido\_id;  END;  // DELIMITER ;  -- Este procedimiento se encarga de insertar un producto nuevo a una factura. Si ya existe pero la cantidad es distinta, se actualizará la fila existente.  DELIMITER //  CREATE PROCEDURE agregar\_producto\_pedido(pedido\_id INT, producto\_id INT, estilo\_id INT, formato\_id INT, cantidad INT, descuento DECIMAL(5,2))  BEGIN      IF ((producto\_id, estilo\_id, formato\_id, descuento) IN (SELECT dp.producto\_id, dp.estilo\_id, dp.formato\_id, dp.descuento FROM detalle\_pedidos dp WHERE dp.pedido\_id = pedido\_id)) THEN          UPDATE detalle\_pedidos dp SET dp.cantidad = dp.cantidad + cantidad WHERE dp.pedido\_id = pedido\_id AND dp.producto\_id = producto\_id AND dp.estilo\_id = estilo\_id AND dp.formato\_id = formato\_id;      ELSE          INSERT INTO detalle\_pedidos (pedido\_id, producto\_id, estilo\_id, formato\_id, cantidad, descuento) VALUES          (pedido\_id, producto\_id, estilo\_id, formato\_id, cantidad, descuento);      END IF;  END;  // DELIMITER ;  -- Este procedimiento muestra todas las facturas de un cliente dado.  DELIMITER //  CREATE PROCEDURE listar\_facturas\_cliente(cliente\_id INT)  BEGIN      SELECT f.\*, ep.nombre estado FROM facturas f JOIN pedidos pe ON f.pedido\_id = pe.id JOIN clientes c ON c.id = pe.cliente\_id JOIN estado\_pedido ep ON ep.id = pe.estado\_id WHERE c.id = cliente\_id;  END;  // DELIMITER ;  -- Este procedimiento actualiza un pedido a su siguiente estado siguiendo el orden logístico (Excepto al estado 'Entregado' y 'Cancelado').  DELIMITER //  CREATE PROCEDURE actualizar\_siguiente\_estado\_pedido(pedido\_id INT)  BEGIN      IF ((SELECT pe.estado\_id FROM pedidos pe WHERE pe.id = pedido\_id) + 1 < 4) THEN          UPDATE pedidos pe SET pe.estado\_id = pe.estado\_id + 1 WHERE pe.id = pedido\_id;      END IF;  END;  // DELIMITER ;  -- Este procedimiento se encarga de establecer el estado de un pedido dado a 'Entregado'.  DELIMITER //  CREATE PROCEDURE entregar\_pedido(pedido\_id INT)  BEGIN      UPDATE pedidos SET estado\_id = 4 WHERE id = pedido\_id;  END;  // DELIMITER ;  -- Este procedimiento se encarga de cancelar un pedido dado, estableciendo su estado a 'Cancelado'.  DELIMITER //  CREATE PROCEDURE cancelar\_pedido(pedido\_id INT)  BEGIN      UPDATE pedidos SET estado\_id = 5 WHERE id = pedido\_id;  END;  // DELIMITER ;  -- Este procedimiento muestra todas las actualizaciones de un pedido que se pasa por parámetro.  DELIMITER //  CREATE PROCEDURE ver\_actualizaciones\_estado\_pedido(pedido\_id INT)  BEGIN      SELECT \* FROM seguimiento\_pedidos s WHERE s.pedido\_id = pedido\_id;  END;  // DELIMITER ;  -- Este procedimiento muestra todos los trabajadores que tiene un local  DELIMITER //  CREATE PROCEDURE ver\_empleados\_local(local\_id INT)  BEGIN      SELECT e.nombre, e.apellidos, t.nombre tipo, l.direccion, l.localidad, l.email, l.telefono FROM empleados e JOIN local l ON l.id = e.local\_id JOIN tipo\_local t ON t.id = l.tipo\_id WHERE l.id = local\_id;  END;  // DELIMITER ;  -- Este procedimiento se encarga de eliminar todos los datos de un cliente, incluidos sus pedidos y los detalles de cada pedido.  DELIMITER //  CREATE PROCEDURE eliminar\_cliente\_completo(cliente\_id INT)  BEGIN      DELETE FROM detalle\_pedidos dp WHERE dp.pedido\_id IN (SELECT pe.id FROM pedidos pe WHERE pe.cliente\_id = cliente\_id);      DELETE FROM pedidos pe WHERE pe.cliente\_id = cliente\_id;      DELETE FROM clientes WHERE id = cliente\_id;  END;  // DELIMITER ; |

# Triggers

|  |
| --- |
| USE card\_customs;  -- Este trigger se asegura de que siempre se use el precio correcto al insertar un producto en la tabla 'detalle\_pedidos'.  DELIMITER //  CREATE TRIGGER insertar\_precio\_unitario\_detalle\_pedidos  BEFORE INSERT ON detalle\_pedidos  FOR EACH ROW  BEGIN      SET NEW.precio\_unitario = (SELECT p.precio FROM productos p WHERE id = NEW.producto\_id);  END;  // DELIMITER ;  -- Este trigger se encarga de comprobar y actualizar el stock antes de insertar un nuevo producto en la tabla 'detalle\_pedidos'.  DELIMITER //  CREATE TRIGGER comprobar\_actualizar\_stock\_detalle\_pedido\_INSERT  BEFORE INSERT ON detalle\_pedidos  FOR EACH ROW  BEGIN      DECLARE diferencia INT DEFAULT (obtener\_stock\_producto(NEW.producto\_id) - NEW.cantidad);        IF (diferencia < 0) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No hay suficiente stock';      ELSE          UPDATE productos SET stock = stock - NEW.cantidad WHERE id = NEW.producto\_id;      END IF;  END;  // DELIMITER ;  -- Este trigger se encarga de comprobar y actualizar el stock en la tabla 'productos' cuando se actualiza una fila en la tabla 'detalle\_pedidos'.  DELIMITER //  CREATE TRIGGER comprobar\_actualizar\_stock\_detalle\_pedido\_UPDATE  BEFORE UPDATE ON detalle\_pedidos  FOR EACH ROW  BEGIN      IF (NEW.cantidad > OLD.cantidad) THEN          IF (obtener\_stock\_producto(OLD.producto\_id) - (NEW.cantidad - OLD.cantidad) < 0) THEN              SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No hay suficiente stock';          END IF;          UPDATE productos SET stock = stock - (NEW.cantidad - OLD.cantidad) WHERE id = OLD.producto\_id;      ELSEIF (NEW.cantidad < OLD.cantidad) THEN          UPDATE productos SET stock = stock + (OLD.cantidad - NEW.cantidad) WHERE id = OLD.producto\_id;      END IF;  END;  // DELIMITER ;  -- Este trigger registra los pedidos que ya hayan finalizado todo su proceso logístico. Estos son considerados como los que su estado es 'Entregado' o 'Cancelado'.  DELIMITER //  CREATE TRIGGER registrar\_pedidos\_historial  AFTER UPDATE ON pedidos  FOR EACH ROW  BEGIN      IF (NEW.estado\_id IN (4, 5)) THEN          INSERT INTO historial\_pedidos (pedido\_id, fecha\_registro, estado\_final) VALUES (NEW.id, now(), NEW.estado\_id);      END IF;  END;  // DELIMITER ;  -- Este trigger comprueba que durante el proceso de actualización de estado de un pedido, no puedan haber saltos entre los estados y se asegura de que se siga un proceso logístico correcto.  DELIMITER //  CREATE TRIGGER comprobar\_proceso\_logistico  AFTER UPDATE ON pedidos  FOR EACH ROW  BEGIN      IF (OLD.estado\_id = 1 AND NEW.estado\_id = 3) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'El estado no puede pasar directamente de "Pendiente" a "Enviado"';      ELSEIF (OLD.estado\_id = 1 AND NEW.estado\_id = 4) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'El estado no puede pasar directamente de "Pendiente" a "Entregado"';      ELSEIF (OLD.estado\_id = 2 AND NEW.estado\_id = 1) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No se puede volver del estado "En preparación" al estado "Pendiente"';      ELSEIF (OLD.estado\_id = 2 AND NEW.estado\_id = 4) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'El estado no puede pasar directamente de "En preparación" a "Entregado"';      ELSEIF (OLD.estado\_id = 3 AND NEW.estado\_id = 1) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No se puede volver del estado "Enviado" al estado "Pendiente"';      ELSEIF (OLD.estado\_id = 3 AND NEW.estado\_id = 2) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No se puede volver del estado "Enviado" al estado "En preparación"';      ELSEIF (OLD.estado\_id IN (4, 5) AND NEW.estado\_id IN (1, 2, 3, 4, 5)) THEN          SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'El pedido ha llegado a su estado final y no puede ser modificado';      END IF;  END;  // DELIMITER ;  -- Este trigger registra cualquier pedido insertado en la tabla 'pedidos'.  DELIMITER //  CREATE TRIGGER registrar\_seguimiento\_inicial\_pedidos  AFTER INSERT ON pedidos  FOR EACH ROW  BEGIN      INSERT INTO seguimiento\_pedidos (pedido\_id, fecha\_registro, estado\_anterior, estado\_nuevo) VALUES (NEW.id, now(), NULL, obtener\_nombre\_estado(NEW.estado\_id));  END;  // DELIMITER ;  -- Este trigger se encarga de registrar cualquier cambio de estado de un pedido.  DELIMITER //  CREATE TRIGGER registrar\_seguimiento\_pedidos  AFTER UPDATE ON pedidos  FOR EACH ROW  BEGIN      INSERT INTO seguimiento\_pedidos (pedido\_id, fecha\_registro, estado\_anterior, estado\_nuevo) VALUES (OLD.id, now(), obtener\_nombre\_estado(OLD.estado\_id), obtener\_nombre\_estado(NEW.estado\_id));  END;  // DELIMITER ; |

# Consultas SQL

|  |
| --- |
| USE card\_customs;  -- Esta consulta funciona como una alerta de stock, clasificandolo por la cantidad de stock que hay.  SELECT p.id, p.nombre, p.stock,  CASE      WHEN p.stock > 10000 THEN 'Muy Alto'      WHEN p.stock BETWEEN 5000 AND 10000 THEN 'Alto'      WHEN p.stock BETWEEN 2500 AND 5000 THEN 'Normal'      WHEN p.stock BETWEEN 1000 AND 2500 THEN 'Bajo'      WHEN p.stock BETWEEN 0 AND 1000 THEN 'Muy bajo'      ELSE 'Sin stock'  END estado  FROM productos p  ORDER BY p.stock ASC;  -- Devuelve todos los pedidos que se encuentran en estado "Pendiente" o "En preparación", mostrando la ID del pedido, la fecha, y el nombre del cliente.  SELECT pe.id, pe.fecha\_pedido, c.nombre, c.apellidos, ep.nombre estado  FROM pedidos pe      JOIN clientes c ON pe.cliente\_id = c.id      JOIN estado\_pedido ep ON pe.estado\_id = ep.id  WHERE ep.nombre IN ('Pendiente', 'En preparación')  ORDER BY pe.fecha\_pedido ASC;  -- Devuelve el precio total de ventas de cada estilo de carta. Algo así como para saber que estilos son más populares o más vendidos.  SELECT e.nombre, concat(sum(dp.cantidad \* dp.precio\_unitario), ' €') total\_ventas  FROM detalle\_pedidos dp      JOIN estilos e ON dp.estilo\_id = e.id  GROUP BY e.nombre  ORDER BY sum(dp.cantidad \* dp.precio\_unitario) DESC;  -- Muestra los datos completos de cada empleado junto con el tipo de local donde trabajan.  SELECT e.nombre, e.apellidos, e.email, l.direccion, t.nombre  FROM empleados e      JOIN local l ON e.local\_id = l.id      JOIN tipo\_local t ON l.tipo\_id = t.id;  -- Lista los clientes que han hecho algún pedido en el ultimo mes.  SELECT DISTINCT c.\*  FROM clientes c      JOIN pedidos p ON c.id = p.cliente\_id  WHERE p.fecha\_pedido >= date\_sub(curdate(), INTERVAL 1 MONTH);  -- Cuenta el numero de locales de tipo 'Tienda' que hay.  SELECT t.nombre, count(\*) cantidad  FROM local l      JOIN tipo\_local t ON t.id = l.tipo\_id  WHERE t.nombre LIKE 'tienda'  GROUP BY t.nombre;  -- Cuenta el numero de locales que hay de cada tipo en una localidad dada  SELECT t.nombre, count(\*) cantidad  FROM local l      JOIN tipo\_local t ON t.id = l.tipo\_id  WHERE l.localidad LIKE 'Madrid'  GROUP BY t.nombre;  -- Devuelve el empleado y el local donde trabaja el empleado con más sueldo en la empresa.  SELECT e.nombre, e.apellidos, t.nombre, l.localidad, l.direccion, e.salario  FROM empleados e      JOIN local l ON l.id = e.local\_id      JOIN tipo\_local t ON t.id = l.tipo\_id  WHERE e.salario = mayor\_salario\_empleados();  -- Devuelve los gastos en sueldos de cada tienda filtrado por su localidad.  SELECT l.localidad, concat(sum(e.salario), ' €') gastos\_sueldos  FROM empleados e      JOIN local l ON e.local\_id = l.id  GROUP BY l.localidad  ORDER BY sum(e.salario) DESC;  -- Devuelve el total ganado con todas las ventas de pedidos que se hayan entregado y no se hayan cancelado.  SELECT concat(sum(f.total), ' €') total\_ventas  FROM facturas f      JOIN pedidos\_finalizados pf ON f.pedido\_id = pf.pedido\_id  WHERE pf.estado\_final LIKE 'Entregado';  -- Devuelve los clientes que hayan gastado más de 100€ en total en todos sus pedidos que no hayan sido cancelados.  SELECT c.nombre, c.apellidos, c.email, concat(sum(f.total), ' €') total\_gastado, count(p.id) cantidad\_pedidos  FROM clientes c      JOIN pedidos p ON p.cliente\_id = c.id      JOIN facturas f ON f.pedido\_id = p.id      JOIN pedidos\_finalizados pf ON pf.pedido\_id = f.pedido\_id  WHERE pf.estado\_final LIKE 'Entregado'  GROUP BY c.id  HAVING sum(f.total) > 100  ORDER BY sum(f.total) DESC;  -- Devuelve los empleados que cobren más que la media de los salarios.  SELECT e.dni, e.nombre, e.apellidos, e.salario, round((SELECT avg(em.salario) FROM empleados em), 2) media\_salario  FROM empleados e  WHERE salario > (SELECT avg(em.salario) FROM empleados em)  ORDER BY e.salario DESC; |

# Casos de Prueba y Simulación

Consideraciones Importantes:

# Resultados y Verificación

Describe aquí...

# Capturas de Pantalla (opcional)

Describe aquí...

# Conclusiones y Mejoras Futuras

Este proyecto ha logrado establecer un sistema de base de datos robusto y funcional para **Card Customs**, modelando eficientemente sus operaciones clave. Se ha implementado un esquema relacional completo, y la lógica de negocio esencial, como cálculos de facturación y gestión de stock, se ha implementado mediante funciones, procedimientos y triggers.

Este sistema no solo satisface las necesidades operativas del negocio, sino que también sirve como una sólida base para futuras expansiones y optimizaciones, demostrando la viabilidad y el valor de una gestión de datos eficiente.

Algunas de esas futuras mejoras que se pueden aplicar a esta base de datos son:

* Establecer un sistema para gestionar las **compras internas a proovedores** que se realicen en la empresa.
* Controlar de manera más detallada el **reparto de productos** desde los almacenes a las tiendas.
* Agregar la posibilidad de tener **diferentes productos** (sin necesidad de ser sólo cartas).
* Añadir **códigos promocionales** para descuentos.
* Añadir la posibilidad tener más direcciones de entrega.
* Agregar la opción de que cada empleado tenga un rol o puesto dentro de la empresa.

# Enlace al Repositorio en GitHub

<https://github.com/jCanay/card-customs>