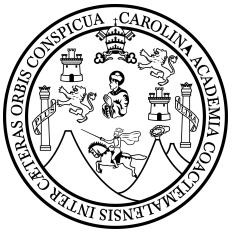
Ingeniería en Ciencias y Sistemas
División de Ciencias de la Ingeniería
Centro Universitario de Occidente
Universidad de San Carlos de Guatemala
Manejo e implementación de archivos



Manual Tecnico
Proyecto 1 - GameProXela

Byron Javier Vasquez Villagran
201931806
#20

Índice

Tecnologías utilizadas..... 3
Script de creación de base de datos.....4
Explicación de las Tablas.....8
Declaracion de triggers..... 11
Explicación de los Triggers.....14

Tecnologías utilizadas

Framework: Laravel

- **Versión**: Laravel 11.25
- **Dependencias**:
    - PHP >= 8.1
    - Extensiones de PHP requeridas: PDO

Base de Datos: PostgreSQL

- **Versión**: PostgreSQL 16.4

Entorno de Desarrollo: Visual Studio Code

- **Versión**: Visual Studio Code 1.93.1

Gestor de Bases de Datos: DataGrip

- **Versión**: DataGrip 2024.2.2

Sistema Operativo: Windows

- **Versión**: Windows 11

Script de creación de base de datos

Creamos la base de datos

```
DROP DATABASE IF EXISTS gamerproxela;
CREATE DATABASE gamerproxela;

CREATE SCHEMA sales_schema;
CREATE SCHEMA stores_schema;
CREATE SCHEMA customers_schema;
```

Creamos las tablas de la base de datos

```
CREATE TABLE sales_schema.roles
(
   uid     SERIAL PRIMARY KEY,
   rolname VARCHAR(50) NOT NULL
);

CREATE TABLE sales_schema.users
(
   uid      SERIAL,
   dpi      BIGINT      NOT NULL
       CONSTRAINT users_pk
           PRIMARY KEY,
   username VARCHAR(20) NOT NULL,
   password VARCHAR(65) NOT NULL,
   rol      INTEGER
       CONSTRAINT rol_fkey
           REFERENCES sales_schema.roles (uid),
   active   INTEGER DEFAULT 1
);

CREATE TABLE customers_schema.customers
(
   uid     SERIAL,
   nit     BIGINT       NOT NULL
       CONSTRAINT customers_pk
           PRIMARY KEY,
   name    VARCHAR(50)  NOT NULL,
   address VARCHAR(100) NOT NULL,
   dpi     BIGINT       NOT NULL
       constraint customers_users_dpi_fk
           REFERENCES sales_schema.users

);


CREATE TABLE customers_schema.memberships
(
   uid         SERIAL
       CONSTRAINT memberships_pk
```

```
            PRIMARY KEY,
   name         VARCHAR NOT NULL,
   requirement BIGINT NOT NULL,
   off_percent INTEGER DEFAULT 0
);
```

Ingresamos los valores por defecto, en este caso las membresias

```
INSERT INTO customers_schema.memberships(name, off_percent, requirement)
VALUES ('comun', 5, 0),
      ('oro', 10, 10000),
      ('platino', 20, 20000),
      ('diamante', 30, 30000);


CREATE TABLE customers_schema.cards
(
   uid             SERIAL
       CONSTRAINT card_pk
           PRIMARY KEY,
   customer_nit    BIGINT
       CONSTRAINT memberships_customer_nit_fk
           REFERENCES customers_schema.customers (nit),
   activation_date DATE    DEFAULT CURRENT_DATE,
   points FLOAT NOT NULL DEFAULT 0,
   member_grade    INTEGER DEFAULT 1
       CONSTRAINT member_grade_fk
           REFERENCES customers_schema.memberships (uid)
);

CREATE TABLE stores_schema.stores
(
   uid     BIGINT
       CONSTRAINT store_pk
           PRIMARY KEY,
   name    VARCHAR(50) NOT NULL,
   phone   BIGINT      NOT NULL,
   address VARCHAR(50) NOT NULL
);

CREATE TABLE sales_schema.employees
(
   uid           SERIAL,
   user_dpi      BIGINT
       CONSTRAINT employer_user_fkey
           REFERENCES sales_schema.users (dpi)
       PRIMARY KEY,
   store_employe BIGINT
       CONSTRAINT employer_store_uid_fkey
           REFERENCES stores_schema.stores (uid),

   name          VARCHAR(50) NOT NULL,
   forename      VARCHAR(50) NOT NULL,
```

```
   no_cashr      INTEGER     NOT NULL DEFAULT 0,
   birthday      DATE        NOT NULL
);


CREATE TABLE stores_schema.products
(
   uid          SERIAL
       CONSTRAINT product_pk
           PRIMARY KEY,
   name         VARCHAR(50) NOT NULL,
   description  VARCHAR(100) DEFAULT ('NO DESC'),
   quantity     INT         NOT NULL,
   cost         FLOAT       NOT NULL,
   public_price FLOAT       NOT NULL
);

CREATE TABLE stores_schema.ledge
(
   store_uid   BIGINT
       CONSTRAINT store_uid_fk
           REFERENCES stores_schema.stores (uid),
   product_uid INTEGER
       CONSTRAINT product_uid_fk
           REFERENCES stores_schema.products (uid),
   quantity    INTEGER NOT NULL,
   no_hall     INTEGER NOT NULL,
   PRIMARY KEY (store_uid, product_uid)
);

CREATE TABLE stores_schema.inventory
(
   store_uid   BIGINT
       CONSTRAINT store_uid_fk
           REFERENCES stores_schema.stores (uid),
   product_uid INTEGER
       CONSTRAINT product_uid_fk
           REFERENCES stores_schema.products (uid),
   quantity    INTEGER NOT NULL
);

CREATE TABLE sales_schema.sales
(
   uid          SERIAL
       CONSTRAINT sale_fk
           PRIMARY KEY,
   customer_nit BIGINT
       CONSTRAINT sale_customer_nit_fkey
           REFERENCES customers_schema.customers (nit),
   store_uid    BIGINT
       CONSTRAINT sale_store_uid_fkey
           REFERENCES stores_schema.stores (uid),
   employee_dpi BIGINT
```

```
       CONSTRAINT sale_employee_dpi_fkey
           REFERENCES sales_schema.employees (user_dpi),
   off_percent  FLOAT NOT NULL DEFAULT 0.0,
   total        FLOAT          DEFAULT 0.0 NOT NULL,
   sale_date    DATE           DEFAULT CURRENT_DATE
);

CREATE TABLE sales_schema.product_list
(
   sale_uid    INTEGER
       CONSTRAINT product_list_sale_uid_fkey
           REFERENCES sales_schema.sales (uid),
   product_uid INTEGER
       CONSTRAINT product_lists_product_uid_fkey
           REFERENCES stores_schema.products (uid),
   quantity    INTEGER NOT NULL,
   subtotal    FLOAT   NOT NULL
);

INSERT INTO sales_schema.roles(rolname)
values ('customer'),
      ('checker'),
      ('storer'),
      ('inventorier');
```

Explicación de las Tablas

1. sales_schema.roles
    - Contiene los roles que los usuarios pueden tener dentro del sistema.
    - Campos:
        - uid: Identificador único del rol (clave primaria).
        - rolname: Nombre del rol (obligatorio).
2. sales_schema.users
    - Almacena la información de los usuarios del sistema.
    - Campos:
        - uid: Identificador único del usuario (se genera automáticamente).
        - dpi: Número de DPI del usuario (clave primaria).
        - username: Nombre de usuario (obligatorio).
        - password: Contraseña del usuario (obligatorio).
        - rol: Referencia al rol del usuario (uid de roles).
        - active: Indica si el usuario está activo (1 por defecto).
3. customers_schema.customers
    - Almacena los clientes registrados en el sistema.
    - Campos:
        - uid: Identificador único del cliente (se genera automáticamente).
        - nit: NIT del cliente (clave primaria).
        - name: Nombre del cliente (obligatorio).
        - address: Dirección del cliente (obligatorio).
        - dpi: Referencia al DPI del usuario en sales_schema.users.
4. customers_schema.memberships
    - Define los tipos de membresías disponibles para los clientes.
    - Campos:
        - uid: Identificador único de la membresía (clave primaria).
        - name: Nombre de la membresía (obligatorio).
        - requirement: Requisito en puntos o cantidad para obtener la membresía.
        - off_percent: Descuento en porcentaje otorgado por la membresía.
5. customers_schema.cards
    - Relaciona clientes con tarjetas de membresía.
    - Campos:
        - uid: Identificador único de la tarjeta (clave primaria).
        - customer_nit: Referencia al NIT del cliente en customers_schema.customers.
        - activation_date: Fecha de activación de la tarjeta (por defecto, la fecha actual).
        - points: Puntos acumulados en la tarjeta (0 por defecto).
        - member_grade: Referencia al nivel de membresía en customers_schema.memberships.
6. stores_schema.stores

- Almacena la información de las tiendas físicas del sistema.
- Campos:
  - uid: Identificador único de la tienda (clave primaria).
  - name: Nombre de la tienda (obligatorio).
  - phone: Número de teléfono de la tienda (obligatorio).
  - address: Dirección de la tienda (obligatorio).

7. sales_schema.employees
   - Registra a los empleados que trabajan en las tiendas.
   - Campos:
     - uid: Identificador único del empleado (se genera automáticamente).
     - user_dpi: Referencia al DPI del usuario en sales_schema.users (clave primaria).
     - store_employe: Referencia al identificador de la tienda en stores_schema.stores.
     - name: Nombre del empleado (obligatorio).
     - forename: Apellido del empleado (obligatorio).
     - no_cashr: Número de cajero (por defecto, 0).
     - birthday: Fecha de nacimiento del empleado (obligatorio).
8. stores_schema.products
   - Contiene la información de los productos que se venden en las tiendas.
   - Campos:
     - uid: Identificador único del producto (clave primaria).
     - name: Nombre del producto (obligatorio).
     - description: Descripción del producto (opcional, "NO DESC" por defecto).
     - quantity: Cantidad disponible del producto (obligatorio).
     - cost: Costo del producto (obligatorio).
     - public_price: Precio de venta al público del producto (obligatorio).
9. stores_schema.ledge
   - Registra la entrada o salida de productos en las tiendas.
   - Campos:
     - store_uid: Referencia a la tienda en stores_schema.stores (parte de la clave primaria).
     - product_uid: Referencia al producto en stores_schema.products (parte de la clave primaria).
     - quantity: Cantidad de productos involucrados en la transacción.
     - no_hall: Número de sala o pasillo donde se encuentra el producto.
10. stores_schema.inventory
    - Lleva un control del inventario de productos en las tiendas.
    - Campos:
      - store_uid: Referencia a la tienda en stores_schema.stores.
      - product_uid: Referencia al producto en stores_schema.products.
      - quantity: Cantidad disponible del producto en la tienda.

11. sales_schema.sales
    - Registra las ventas realizadas en el sistema.
    - Campos:
        - uid: Identificador único de la venta (clave primaria).
        - customer_nit: Referencia al cliente en customers_schema.customers.
        - store_uid: Referencia a la tienda en stores_schema.stores.
        - employee_dpi: Referencia al empleado que realizó la venta en sales_schema.employees.
        - off_percent: Porcentaje de descuento aplicado en la venta (por defecto, 0%).
        - total: Total de la venta (obligatorio).
        - sale_date: Fecha de la venta (por defecto, la fecha actual).
12. sales_schema.product_list
    - Lista los productos involucrados en una venta.
    - Campos:
        - sale_uid: Referencia a la venta en sales_schema.sales.
        - product_uid: Referencia al producto en stores_schema.products.
        - quantity: Cantidad del producto en la venta.
        - subtotal: Subtotal correspondiente a la cantidad de producto.
13. Inserciones en customers_schema.memberships
    - Se insertan diferentes niveles de membresía: "comun", "oro", "platino", y "diamante", con sus respectivos descuentos y requisitos de puntos.
14. Inserciones en sales_schema.roles
    - Se insertan roles iniciales: "customer", "checker", "storer", y "inventorier".

Declaracion de triggers

```
CREATE OR REPLACE FUNCTION addPointsToCard()
   RETURNS TRIGGER AS
$$
DECLARE
   customer_nit   BIGINT;
   member_grade   INTEGER;
   points_to_add  INTEGER;
   earning_points INTEGER;
BEGIN
   SELECT c.customer_nit as nit, c.member_grade, m.off_percent
   INTO customer_nit, member_grade, earning_points
   FROM customers_schema.cards c
            left join customers_schema.memberships m on c.member_grade =
m.uid
   WHERE c.customer_nit = NEW.customer_nit;


   points_to_add := FLOOR(NEW.total / 200) * earning_points;

   UPDATE customers_schema.cards c
   SET points = points + points_to_add
   WHERE c.customer_nit = NEW.customer_nit;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_sale_insert
   AFTER INSERT
   ON sales_schema.sales
   FOR EACH ROW
EXECUTE FUNCTION addPointsToCard();

CREATE OR REPLACE FUNCTION updateInventory()
   RETURNS TRIGGER AS
$$
BEGIN
   UPDATE stores_schema.inventory
   SET quantity = quantity - NEW.quantity
   WHERE store_uid = (SELECT store_uid FROM sales_schema.sales WHERE uid =
NEW.sale_uid)
     AND product_uid = NEW.product_uid;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_product_list_insert
   AFTER INSERT
   ON sales_schema.product_list
```

```
   FOR EACH ROW
EXECUTE FUNCTION updateInventory();

CREATE OR REPLACE FUNCTION upgradeLevelCard()
   RETURNS TRIGGER AS
$$
DECLARE
   new_grade INTEGER;
BEGIN

   IF NEW.member_grade >= (SELECT requirement FROM
customers_schema.memberships where name = 'diamante') THEN
       new_grade := (SELECT uid FROM customers_schema.memberships WHERE name
= 'diamante');
   ELSIF NEW.member_grade >= (SELECT requirement FROM
customers_schema.memberships where name = 'platino') THEN
       new_grade := (SELECT uid FROM customers_schema.memberships WHERE name
= 'platino');
   ELSIF NEW.member_grade >= (SELECT requirement FROM
customers_schema.memberships where name = 'oro') THEN
       new_grade := (SELECT uid FROM customers_schema.memberships WHERE name
= 'oro');
   ELSE
       new_grade := (SELECT uid FROM customers_schema.memberships WHERE name
= 'comun');
   END IF;

   IF NEW.member_grade <> new_grade THEN
       UPDATE customers_schema.cards
       SET member_grade = new_grade
       WHERE customer_nit = NEW.customer_nit;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_card_points_update
   AFTER UPDATE OF member_grade
   ON customers_schema.cards
   FOR EACH ROW
EXECUTE FUNCTION upgradeLevelCard();


CREATE OR REPLACE FUNCTION comprobarexistencia()
   RETURNS TRIGGER
   LANGUAGE plpgsql
AS
$$
DECLARE
   ex INTEGER;
BEGIN
   SELECT quantity
```

```
   INTO ex
   FROM stores_schema.inventory i
   WHERE store_uid = (SELECT store_uid FROM sales_schema.sales WHERE uid =
NEW.sale_uid)
     AND i.product_uid = NEW.product_uid;


   IF NEW.quantity > ex THEN
       RAISE EXCEPTION 'CANTIDAD INSUFICIENTE, EXISTENCIA ACTUAL: %', ex;
   END IF;

   RETURN NEW;
END
$$;

CREATE TRIGGER before_product_list_insert
   BEFORE INSERT
   ON sales_schema.product_list
   FOR EACH ROW
EXECUTE FUNCTION comprobarexistencia();
```

Explicación de los Triggers

1. Trigger: after_sale_insert
    - Función asociada: addPointsToCard
    - Descripción: Este trigger se activa después de que se inserta un nuevo registro en la tabla sales_schema.sales. La función addPointsToCard agrega puntos a la tarjeta del cliente asociada a la venta, basado en el total de la venta y el nivel de membresía del cliente.
    - Detalles de la función:
        - Se selecciona el NIT del cliente (customer_nit), su nivel de membresía (member_grade), y el porcentaje de descuento otorgado por la membresía (off_percent).
        - Los puntos a agregar se calculan dividiendo el total de la venta por 200 y multiplicándolo por el porcentaje de puntos obtenidos según la membresía.
        - Finalmente, se actualizan los puntos de la tarjeta del cliente.
2. Trigger: after_product_list_insert
    - Función asociada: updateInventory
    - Descripción: Se activa después de que se inserta un registro en la tabla sales_schema.product_list. La función updateInventory actualiza la cantidad del producto en el inventario después de que se ha registrado una venta de productos.
    - Detalles de la función:
        - Se actualiza la cantidad disponible del producto en el inventario, restando la cantidad vendida del producto asociado a la tienda correspondiente.
3. Trigger: after_card_points_update
    - Función asociada: upgradeLevelCard
    - Descripción: Se activa después de que se actualiza el campo member_grade en la tabla customers_schema.cards. La función upgradeLevelCard verifica si los puntos acumulados del cliente cumplen con los requisitos para un nivel de membresía más alto, y actualiza el nivel de membresía si es necesario.
    - Detalles de la función:
        - Se evalúa el nivel de membresía del cliente basándose en los puntos acumulados (member_grade) y se asigna el nuevo nivel si es superior al actual.
        - Los niveles de membresía se verifican en orden: diamante, platino, oro, común.
        - Si el nivel de membresía ha cambiado, se actualiza la tarjeta del cliente con el nuevo nivel.
4. Trigger: before_product_list_insert
    - Función asociada: comprobarexistencia

- Descripción: Se activa antes de insertar un nuevo registro en la tabla sales_schema.product_list. La función comprobarexistencia verifica si hay suficiente cantidad del producto en el inventario antes de permitir la venta.
- Detalles de la función:
  - Se consulta el inventario para obtener la cantidad disponible del producto en la tienda correspondiente.
  - Si la cantidad solicitada en la venta excede la cantidad disponible, se lanza una excepción que notifica la cantidad actual disponible.
  - De esta forma, se impide que se procese una venta con productos insuficientes en el inventario.