**Documentação de modelagem de dados – Olimpo**

**Introdução:**

Para documentar as diferenças entre os scripts dos bancos de dados do primeiro e segundo ano, este documento apresenta uma visão clara e organizada das principais distinções. Modelagem de dados é o processo de estruturar e organizar informações de maneira lógica e eficiente dentro de um banco de dados, facilitando o acesso e a manipulação dos dados. Aqui, é fornecida a explicação de como as triggers, procedures e functions foram implementadas e ajustadas no banco do segundo ano, destacando suas funcionalidades específicas e evoluções ao longo do tempo.

**Tabelas do banco do primeiro ano:**

* **Categoria**: Armazena as categorias de anúncios, com um nome único e atributos para indicar se foi atualizada ou excluída.
* **Interesse**: Contém interesses com um nome único, também com indicadores de atualização e exclusão.
* **Cliente**: Registra os dados dos clientes, incluindo email, senha, CPF e um interesse vinculado à tabela Interesse.
* **Endereco**: Armazena os endereços dos clientes, vinculados à tabela Cliente por meio do ID do cliente.
* **Telefone Cliente**: Registra os telefones dos clientes, também com uma ligação ao ID do cliente em Cliente.
* **Plano**: Contém os tipos de planos oferecidos, com nome, valor e indicadores de atualização e exclusão.
* **Anúncio**: Armazena os anúncios criados pelos clientes, com título, descrição, preço, categoria e referência ao cliente e ao plano.
* **Admin**: Contém dados dos administradores do sistema, com nome de usuário, senha e indicadores de atualização e exclusão.

**Tabelas do banco do segundo ano:**

* **Gender**: Armazena os gêneros, com um nome único.
* **Interest**: Contém interesses com um nome único e um valor padrão.
* **Customer**: Registra dados dos clientes, incluindo email, senha, CPF, imagem de perfil e referências para Gender e Interest.
* **Phone\_Customer**: Registra os telefones dos clientes, com vínculo ao ID de cliente em Customer.
* **Address**: Armazena endereços dos clientes, também referenciando a tabela Customer.
* **Category**: Armazena categorias com um nome único.
* **Plan**: Contém tipos de planos com nome único e valor não negativo.
* **Community**: Registra informações sobre comunidades, incluindo nome, data de início e bairro.
* **Advertisement**: Armazena anúncios criados pelos clientes, com título, descrição, preço e referências a Category, Customer e Plan.
* **Announcement**: Registra anúncios dentro das comunidades, incluindo ID da comunidade e dados do remetente.
* **Publication**: Contém publicações feitas pelos clientes, com data de publicação, curtidas e descrição.
* **Community\_Customer**: Relaciona clientes e comunidades, permitindo que clientes participem de várias comunidades.
* **Administrator**: Define administradores dentro das comunidades, vinculando um cliente e uma comunidade.

**Principais diferenças entre os dois bancos:**

**Complexidade Estrutural e Funcionalidade Avançada**:

* O banco de dados do segundo ano possui uma estrutura mais complexa, com tabelas adicionais como Community, Publication, Announcement, e múltiplas tabelas de log (Log\_Customer, Log\_Announcement, etc.), que registram ações de inserção, atualização e exclusão. Isso reflete uma preocupação maior com o monitoramento e histórico das operações.

**Procedures e Funções**:

* O banco de dados do segundo ano contém procedures (insert\_customer, update\_customer, etc.) e funções (check\_administrator) que facilitam a manipulação de dados de forma encapsulada, permitindo operações complexas, como validações de entrada e relacionamento entre dados.

**Triggers para Log de Ações**:

* O banco de dados do segundo ano implementa triggers para registrar logs automáticos em várias tabelas (log\_customer\_action, log\_advertisement\_action, etc.) o que permite uma rastreabilidade detalhada das ações de usuários e alterações de dados.

**Tabelas de Configuração e Constrangimentos**:

* O banco do segundo ano inclui tabelas de configuração como Gender e Interest, com FOREIGN KEY para definir categorias e interesses de forma mais específica, enquanto o banco do primeiro ano lida com essas informações diretamente nas tabelas Cliente e Interesse.

**Campos Padrão e Constrangimentos**:

* O banco do segundo ano usa valores padrão, como interest\_id com valor 'Nenhum', e constrangimentos em campos como email (CHECK (email LIKE '%@%')), trazendo uma camada adicional de validação de dados. No primeiro script, tais restrições são menos comuns.

**Detalhes das tabelas do banco do segundo ano:**

**Por que somente a documentação do banco do segundo ano?**

Após uma análise detalhada dos scripts SQL que compõem o nosso banco de dados, concluímos que a documentação do banco do primeiro ano não é necessária, pois o banco do segundo ano, não só inclui as informações essenciais do primeiro, mas também estabelece relações com outras tabelas, como Gender e Interest, que permitem uma análise mais robusta e contextualizada dos perfis dos usuários. Com isso, a documentação do primeiro script se torna redundante, já que os aspectos fundamentais foram integrados e melhorados no segundo.

**Tabelas do banco:**

**Customer:**

A tabela Customer armazena dados essenciais dos clientes, incluindo identificação pessoal, dados de login, imagem de perfil, e suas preferências de gênero e interesses. Cada coluna representa um campo de informação relevante, com restrições para assegurar a integridade e a exclusividade dos dados.

**Script:**

CREATE TABLE Customer (

ID INT PRIMARY KEY,

email VARCHAR(255) NOT NULL UNIQUE CHECK (email LIKE '%@%'),

password VARCHAR(255) NOT NULL,

name VARCHAR(255) NOT NULL,

surname VARCHAR(255) NOT NULL,

cpf VARCHAR(14) NOT NULL UNIQUE,

profile\_image TEXT,

gender\_id INT,

interest\_id INT DEFAULT 0,

FOREIGN KEY (gender\_id) REFERENCES Gender(ID),

FOREIGN KEY (interest\_id) REFERENCES Interest(ID)

);

**Explicação de cada atributo da tabela:**

* ID (INT): Identificador único para cada cliente, que serve como a chave primária da tabela.
* email (VARCHAR(255)): Endereço de e-mail do cliente, deve ser único e não pode ser nulo; também possui uma verificação para garantir que contém um formato válido de e-mail.
* password (VARCHAR(255)): Senha do cliente, que deve ser obrigatória e não pode ser nula.
* name (VARCHAR(255)): Nome do cliente, um campo obrigatório.
* surname (VARCHAR(255)): Sobrenome do cliente, também um campo obrigatório.
* cpf (VARCHAR(14)): Cadastro de Pessoa Física do cliente, que deve ser único e não pode ser nulo.
* profile\_image (TEXT): Campo para armazenar a imagem de perfil do cliente, que é opcional.
* gender\_id (INT): Referência ao gênero do cliente, que é um campo opcional e vinculado à tabela Gender através de uma chave estrangeira.
* interest\_id (INT, padrão 0): Referência aos interesses do cliente, que é um campo opcional e vinculado à tabela Interest através de uma chave estrangeira.

**Address:**

A tabela Address armazena informações de endereços relacionados aos clientes, incluindo detalhes sobre o bairro, estado e município. Cada endereço é identificado por um ID único e vinculado a um cliente através de uma chave estrangeira.

**Script:**

CREATE TABLE Address (

ID INT PRIMARY KEY,

neighborhood VARCHAR(255),

state VARCHAR(255),

municipality VARCHAR(255),

customer\_id INT,

FOREIGN KEY (customer\_id) REFERENCES Customer(ID)

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada endereço, que serve como a chave primária da tabela.
* **neighborhood (VARCHAR(255)):** Nome do bairro onde o endereço está localizado, um campo opcional.
* **state (VARCHAR(255)):** Nome do estado onde o endereço está localizado, um campo opcional.
* **municipality (VARCHAR(255)):** Nome do município onde o endereço está localizado, um campo opcional.
* **customer\_id (INT):** Referência ao cliente associado a este endereço, que é uma chave estrangeira vinculada à tabela Customer.

**Category**:

A tabela Category armazena as categorias disponíveis para anúncios ou publicações. Cada categoria possui um ID único e um nome que deve ser exclusivo, permitindo a organização eficiente de diferentes tipos de conteúdo.

**Script:**

CREATE TABLE Category (

ID INT PRIMARY KEY,

name VARCHAR(255) NOT NULL UNIQUE

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada categoria, que serve como a chave primária da tabela.
* **name (VARCHAR(255)):** Nome da categoria, um campo obrigatório que deve ser único.

**Plan:**

A tabela Plan armazena informações sobre diferentes planos disponíveis, que podem estar associados a serviços ou produtos. Cada plano é identificado por um ID único e possui um nome e um valor que devem ser exclusivos.

**Script:**

CREATE TABLE Plan (

ID INT PRIMARY KEY,

name VARCHAR(255) NOT NULL UNIQUE,

value FLOAT CHECK (value >= 0)

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada plano, que serve como a chave primária da tabela.
* **name (VARCHAR(255)):** Nome do plano, um campo obrigatório que deve ser único.
* **value (FLOAT):** Valor do plano, que deve ser maior ou igual a zero, garantindo que não haja planos com preços negativos.

**Community:**

A tabela Community armazena informações sobre comunidades, incluindo nome, data de início, bairro e imagem associada. Cada coluna contém dados relevantes que ajudam a caracterizar e identificar a comunidade.

**Script:**

CREATE TABLE Community (

ID INT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

start\_date DATE,

neighborhood VARCHAR(255) DEFAULT 'Não especificado',

image TEXT

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada comunidade, que serve como a chave primária da tabela.
* **name (VARCHAR (255)):** Nome da comunidade, um campo obrigatório.
* **start\_date (DATE):** Data de início da comunidade, um campo opcional.
* **neighborhood (VARCHAR (255)):** Nome do bairro onde a comunidade está localizada, com um valor padrão de 'Não especificado'.
* **image (TEXT):** Campo para armazenar uma imagem representativa da comunidade, que é opcional.

**Advertisement:**

A tabela Advertisement armazena informações sobre anúncios publicados pelos usuários, incluindo detalhes como título, descrição, preço, categoria e a data de publicação. Cada coluna é projetada para capturar dados relevantes sobre os anúncios, com restrições de integridade e chaves estrangeiras que estabelecem relações com outras tabelas.

**Script:**

CREATE TABLE Advertisement (

ID INT PRIMARY KEY,

title VARCHAR(255),

description VARCHAR(255),

publication\_date DATE DEFAULT CURRENT\_DATE,

price FLOAT,

category INT,

image TEXT,

user\_id INT,

plan\_id INT,

FOREIGN KEY (category) REFERENCES Category(ID),

FOREIGN KEY (user\_id) REFERENCES Customer(ID) ON DELETE CASCADE,

FOREIGN KEY (plan\_id) REFERENCES Plan(ID)

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada anúncio, que serve como a chave primária da tabela.
* **title (VARCHAR(255)):** Título do anúncio, um campo opcional que descreve o conteúdo do anúncio.
* **description (VARCHAR(255)):** Descrição do anúncio, também um campo opcional que fornece mais detalhes sobre o que está sendo anunciado.
* **publication\_date (DATE):** Data de publicação do anúncio, que tem um valor padrão de data atual (CURRENT\_DATE).
* **price (FLOAT):** Preço do item ou serviço anunciado, um campo opcional que permite valores de ponto flutuante.
* **category (INT):** Referência à categoria do anúncio, que é um campo opcional e vinculado à tabela Category através de uma chave estrangeira.
* **image (TEXT):** Campo para armazenar a imagem do anúncio, que é opcional.
* **user\_id (INT):** Referência ao usuário que publicou o anúncio, que é um campo obrigatório e vinculado à tabela Customer através de uma chave estrangeira; se o usuário for excluído, os anúncios associados também serão excluídos (ON DELETE CASCADE).
* **plan\_id (INT):** Referência ao plano associado ao anúncio, que é um campo opcional e vinculado à tabela Plan através de uma chave estrangeira.

**Announcement:**

A tabela Announcement armazena informações sobre anúncios feitos em comunidades, incluindo detalhes do remetente, imagem, descrição e tags associadas. Cada coluna é projetada para capturar dados relevantes sobre os anúncios, com restrições de integridade e chaves estrangeiras que estabelecem relações com outras tabelas.

**Script:**

CREATE TABLE Announcement (

ID INT PRIMARY KEY,

community\_id INT,

sender\_id VARCHAR,

sender\_name VARCHAR,

image TEXT,

description VARCHAR,

tag VARCHAR,

FOREIGN KEY (community\_id) REFERENCES Community(ID) ON DELETE CASCADE

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada anúncio, que serve como a chave primária da tabela.
* **community\_id (INT):** Referência à comunidade onde o anúncio foi feito, um campo obrigatório vinculado à tabela Community através de uma chave estrangeira; se a comunidade for excluída, os anúncios associados também serão excluídos (ON DELETE CASCADE).
* **sender\_id (VARCHAR):** Identificador do remetente do anúncio, que permite armazenar informações sobre quem enviou o anúncio.
* **sender\_name (VARCHAR):** Nome do remetente do anúncio, fornecendo uma identificação mais legível para o usuário.
* **image (TEXT):** Campo para armazenar a imagem do anúncio, que é opcional.
* **description (VARCHAR):** Descrição do anúncio, um campo opcional que fornece mais detalhes sobre o que está sendo anunciado.
* **tag (VARCHAR):** Campo para armazenar tags associadas ao anúncio, permitindo categorizar ou marcar o conteúdo de forma a facilitar a busca e organização.

**Publication:**

A tabela Publication armazena informações sobre publicações feitas por clientes, incluindo a data de publicação, quantidade de curtidas, descrição e a referência ao cliente que fez a publicação. As colunas são projetadas para capturar dados essenciais, com restrições que asseguram a integridade referencial.

**Script:**

CREATE TABLE Publication (

ID INT PRIMARY KEY,

publication\_date DATE,

likes INT,

description VARCHAR(255),

customer\_id INT,

FOREIGN KEY (customer\_id) REFERENCES Customer(ID) ON DELETE CASCADE

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada publicação, que serve como a chave primária da tabela.
* **publication\_date (DATE):** Data em que a publicação foi feita, permitindo rastrear quando o conteúdo foi criado.
* **likes (INT):** Contador de curtidas recebidas pela publicação, permitindo medir o engajamento do conteúdo.
* **description (VARCHAR(255)):** Descrição da publicação, um campo opcional que fornece detalhes sobre o conteúdo publicado.
* **customer\_id (INT):** Referência ao cliente que fez a publicação, um campo obrigatório vinculado à tabela Customer através de uma chave estrangeira; se o cliente for excluído, as publicações associadas também serão excluídas (ON DELETE CASCADE).

**Admin:**

A tabela Admin armazena informações sobre administradores do sistema, incluindo suas credenciais de login. As colunas são projetadas para assegurar que cada administrador tenha um identificador único e um conjunto de credenciais para acesso ao sistema.

**Script:**

CREATE TABLE Admin (

ID INT PRIMARY KEY,

username VARCHAR(255),

password VARCHAR(255)

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada administrador, que serve como a chave primária da tabela.
* **username (VARCHAR(255)):** Nome de usuário do administrador, que deve ser único e é utilizado para login no sistema.
* **password (VARCHAR(255)):** Senha do administrador, que é utilizada para autenticação; deve ser armazenada de forma segura.

**Gender:**

A tabela Gender armazena os diferentes gêneros que podem ser associados aos clientes. Cada gênero é identificado por um ID único e um nome que não pode ser duplicado.

**Script:**

CREATE TABLE Gender (

ID INT PRIMARY KEY,

name VARCHAR(20) NOT NULL UNIQUE

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada gênero, que serve como a chave primária da tabela.
* **name (VARCHAR(20)):** Nome do gênero, que deve ser único e não pode ser nulo.

**Interest:**

A tabela Interest armazena diferentes interesses que podem ser associados aos clientes. Cada interesse é identificado por um ID único e um nome que deve ser exclusivo, com um valor padrão definido.

**Script:**

CREATE TABLE Interest (

ID INT PRIMARY KEY,

name VARCHAR(20) NOT NULL UNIQUE DEFAULT 'Nenhum'

);

**Explicação de cada atributo da tabela:**

* **ID (INT):** Identificador único para cada interesse, que serve como a chave primária da tabela.
* **name (VARCHAR(20)):** Nome do interesse, que deve ser único, não pode ser nulo e possui um valor padrão de 'Nenhum'.

**Tabelas de relacionamento:**

**Phone\_Customer:**

A tabela Phone\_Customer armazena informações sobre os números de telefone dos clientes, vinculando cada número a um cliente específico. Essa tabela é útil para manter o controle dos contatos dos clientes em um formato organizado.

**Script:**

CREATE TABLE Phone\_Customer (

ID INT PRIMARY KEY,

phone VARCHAR(15) NOT NULL,

customer\_id INT,

FOREIGN KEY (customer\_id) REFERENCES Customer(ID)

);

**Explicação de cada atributo da tabela:**

* **ID (INT)**: Identificador único para cada registro de telefone, que serve como a chave primária da tabela.
* **phone (VARCHAR(15))**: Número de telefone do cliente, que deve ser obrigatório e não pode ser nulo.
* **customer\_id (INT)**: Referência ao cliente associado ao número de telefone, vinculado à tabela Customer através de uma chave estrangeira.

**Community\_Customer:**

A tabela Community\_Customer armazena a relação entre clientes e comunidades, permitindo que um cliente pertença a várias comunidades e que uma comunidade tenha vários clientes associados. Esse modelo é útil para gerenciar a participação dos clientes em diferentes grupos ou comunidades.

**Script:**

CREATE TABLE Community\_Customer (

ID SERIAL PRIMARY KEY,

customer\_id INT,

community\_id INT,

FOREIGN KEY (customer\_id) REFERENCES Customer(ID) ON DELETE CASCADE,

FOREIGN KEY (community\_id) REFERENCES Community(ID) ON DELETE CASCADE

);

**Explicação de cada atributo da tabela:**

* **ID (SERIAL)**: Identificador único para cada associação de cliente à comunidade, que serve como a chave primária da tabela.
* **customer\_id (INT)**: Referência ao cliente associado à comunidade, vinculado à tabela Customer através de uma chave estrangeira.
* **community\_id (INT)**: Referência à comunidade à qual o cliente pertence, vinculado à tabela Community através de uma chave estrangeira

**Tabelas de Log**

As tabelas de log são estruturas fundamentais em um sistema de banco de dados que registram ações ou eventos que ocorrem dentro da aplicação. Elas são essenciais para monitorar atividades, auditar ações dos usuários, e diagnosticar problemas. Cada entrada na tabela de log captura informações sobre uma ação específica, permitindo uma rastreabilidade completa das alterações no sistema. As tabelas de log podem ajudar a identificar comportamentos suspeitos, fornecer um histórico de alterações e facilitar a análise de dados para melhorias de sistema.

**Estrutura das Tabelas de Log**

A seguir, estão as tabelas de log implementadas para rastrear ações específicas relacionadas a diferentes entidades do sistema:

CREATE TABLE Log\_Customer (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

customer\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Log\_Announcement (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

announcement\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Log\_Publication (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

publication\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Log\_Community (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

community\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Log\_Plan (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

plan\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Log\_Advertisement (

id SERIAL PRIMARY KEY,

action\_type VARCHAR(50),

advertisement\_id INT,

action\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Procedures:**

**O que são procedures?**

Uma **procedure** (ou procedimento armazenado) é um conjunto de instruções SQL que é armazenado no banco de dados e pode ser executado como uma unidade. Ela é projetada para realizar uma tarefa específica, como inserir, atualizar ou excluir dados, e pode aceitar parâmetros de entrada e saída, permitindo que a lógica de negócios seja reutilizada em diferentes partes de uma aplicação. As procedures ajudam a encapsular operações complexas, tornando o código mais organizado e facilitando a manutenção e a segurança, pois podem reduzir a quantidade de SQL que precisa ser enviado do cliente para o servidor.

**Procedures da tabela cliente:**

**Procedimento: insert\_customer**

A procedure insert\_customer é responsável por inserir um novo cliente na tabela Customer. Ela recebe os seguintes parâmetros de entrada: e-mail, senha, nome, sobrenome, CPF, nome do gênero, nome do interesse e imagem de perfil.

**Script:**

CREATE OR REPLACE PROCEDURE insert\_customer(

p\_email VARCHAR,

p\_password VARCHAR,

p\_name VARCHAR,

p\_surname VARCHAR,

p\_cpf VARCHAR,

p\_gender\_name VARCHAR,

p\_interest\_name VARCHAR,

p\_profile\_image TEXT

)

LANGUAGE plpgsql AS $$

DECLARE

v\_gender\_id INT;

v\_interest\_id INT;

v\_new\_id INT;

BEGIN

IF EXISTS (SELECT 1 FROM Customer WHERE cpf = p\_cpf) THEN

RAISE EXCEPTION 'Customer already exists with CPF: %', p\_cpf;

END IF;

SELECT ID INTO v\_gender\_id FROM Gender WHERE name = p\_gender\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Gender not found: %', p\_gender\_name;

END IF;

SELECT ID INTO v\_interest\_id FROM Interest WHERE name = p\_interest\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Interest not found: %', p\_interest\_name;

END IF;

*-- Calcula o próximo ID*

SELECT COALESCE(MAX(ID), 0) + 1 INTO v\_new\_id FROM Customer;

INSERT INTO Customer (ID, email, password, name, surname, cpf, gender\_id, interest\_id, profile\_image)

VALUES (v\_new\_id, p\_email, p\_password, p\_name, p\_surname, p\_cpf, v\_gender\_id, v\_interest\_id, p\_profile\_image);

END;

$$;

1. **Verificação de Existência**: Primeiro, a procedure verifica se já existe um cliente com o mesmo CPF. Se existir, uma exceção é levantada com a mensagem "Customer already exists with CPF".
2. **Busca de Gênero**: Em seguida, tenta obter o ID correspondente ao nome do gênero fornecido. Se o gênero não for encontrado, uma exceção é levantada.
3. **Busca de Interesse**: Similarmente, busca o ID do interesse. Se não for encontrado, uma exceção é levantada.
4. **Cálculo de ID Novo**: A procedure calcula o próximo ID disponível para o novo cliente, pegando o maior ID atual da tabela Customer e somando 1.
5. **Inserção do Cliente**: Por fim, insere um novo registro na tabela Customer com os dados fornecidos e os IDs de gênero e interesse obtidos anteriormente.

**Procedure: update\_customer**

A procedure update\_customer é responsável por atualizar as informações de um cliente na tabela Customer.

Script:

CREATE OR REPLACE PROCEDURE update\_customer(

p\_field VARCHAR,

p\_new\_value VARCHAR,

p\_customer\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

EXECUTE format('UPDATE Customer SET %I = $1 WHERE ID = $2', p\_field)

USING p\_new\_value, p\_customer\_id;

END;

$$;

1. **Parâmetros de Entrada**: Recebe o ID do cliente (p\_customer\_id) e os novos dados a serem atualizados, como email, nome, sobrenome, CPF, nome do gênero e imagem de perfil.
2. **Busca pelo Gênero**: Realiza uma consulta na tabela Gender para obter o ID correspondente ao nome do gênero fornecido. Se não encontrar o gênero, gera uma exceção.
3. **Atualização dos Dados**: Executa uma instrução UPDATE na tabela Customer, atualizando os campos especificados para o cliente cujo ID corresponde a p\_customer\_id.

**Procedure: delete\_customer:**

A procedure delete\_customer é projetada para excluir um cliente da tabela Customer.

**Script:**

CREATE OR REPLACE PROCEDURE delete\_customer(

p\_customer\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

DELETE FROM Customer WHERE ID = p\_customer\_id;

END;

$$;

1. **Parâmetro de Entrada**: Recebe o ID do cliente (p\_customer\_id) que deve ser excluído.
2. **Exclusão**: Executa uma instrução DELETE para remover o registro do cliente com o ID correspondente a p\_customer\_id da tabela Customer.

**Procedures da tabela advertisement**

**Procedure: insert\_advertisement**

A procedure insert\_advertisement é responsável por inserir um novo anúncio na tabela Advertisement.

Script:

CREATE OR REPLACE PROCEDURE insert\_advertisement(

p\_description VARCHAR,

p\_date DATE,

p\_category\_name VARCHAR,

p\_title VARCHAR,

p\_plan\_name VARCHAR,

p\_customer\_cpf VARCHAR,

p\_image TEXT

)

LANGUAGE plpgsql AS $$

DECLARE

v\_category INT;

v\_customer\_id INT;

v\_plan\_id INT;

v\_new\_id INT;

BEGIN

SELECT ID INTO v\_customer\_id FROM Customer WHERE cpf = p\_customer\_cpf;

IF NOT FOUND THEN

RAISE EXCEPTION 'Customer not found with CPF: %', p\_customer\_cpf;

END IF;

SELECT ID INTO v\_category FROM Category WHERE name = p\_category\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Category not found: %', p\_category\_name;

END IF;

SELECT ID INTO v\_plan\_id FROM Plan WHERE name = p\_plan\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Plan not found: %', p\_plan\_name;

END IF;

*-- Calcula o próximo ID*

SELECT COALESCE(MAX(ID), 0) + 1 INTO v\_new\_id FROM Advertisement;

INSERT INTO Advertisement (ID, description, publication\_date, category, title, user\_id, plan\_id, image)

VALUES (v\_new\_id, p\_description, p\_date, v\_category, p\_title, v\_customer\_id, v\_plan\_id, p\_image);

END;

$$;

1. **Parâmetros de Entrada**: Recebe os dados necessários para criar um anúncio, incluindo descrição, data, nome da categoria, título, nome do plano, CPF do cliente e imagem.
2. **Validação do Cliente**: Busca o ID do cliente na tabela Customer usando o CPF fornecido. Se o cliente não for encontrado, gera uma exceção.
3. **Validação da Categoria**: Busca o ID da categoria na tabela Category usando o nome da categoria fornecido. Caso a categoria não exista, gera uma exceção.
4. **Validação do Plano**: Busca o ID do plano na tabela Plan usando o nome do plano fornecido. Se o plano não for encontrado, gera uma exceção.
5. **Cálculo do Novo ID**: Calcula o próximo ID disponível para o novo anúncio, utilizando o maior ID atual da tabela Advertisement.
6. **Inserção do Anúncio**: Executa uma instrução INSERT para adicionar o novo anúncio na tabela Advertisement, preenchendo os campos com os dados fornecidos e os IDs obtidos nas etapas anteriores.

**Procedure: update\_advertisement**

A procedure update\_advertisement é responsável por atualizar um anúncio existente na tabela Advertisement.

**Script:**

CREATE OR REPLACE PROCEDURE update\_advertisement(

p\_field VARCHAR,

p\_new\_value VARCHAR,

p\_advertisement\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

EXECUTE format('UPDATE Advertisement SET %I = $1 WHERE ID = $2', p\_field)

USING p\_new\_value, p\_advertisement\_id;

END;

$$;

1. **Parâmetros de Entrada**: Recebe o ID do anúncio que deve ser atualizado, juntamente com a nova descrição, título e preço.
2. **Atualização do Anúncio**: Executa uma instrução UPDATE na tabela Advertisement, definindo a nova descrição, título e preço para o anúncio correspondente ao ID fornecido.

**Procedure: delete\_advertisement**

A procedure delete\_advertisement tem a função de remover um anúncio da tabela Advertisement.

**Script:**

CREATE OR REPLACE PROCEDURE delete\_advertisement(

p\_advertisement\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

DELETE FROM Advertisement WHERE ID = p\_advertisement\_id;

END;

$$;

1. **Parâmetro de Entrada**: Recebe o ID do anúncio que deve ser excluído.
2. **Exclusão do Anúncio**: Executa uma instrução DELETE na tabela Advertisement, removendo o registro que corresponde ao ID fornecido.

**Procedures da tabela announcement:**

**Procedure: insert\_announcement**

A procedure insert\_announcement é responsável por adicionar um novo anúncio à tabela Announcement.

Script:

1. **Parâmetros de Entrada**: Recebe os detalhes do anúncio, incluindo o ID da comunidade, o ID e o nome do remetente, a imagem, a descrição e a tag.
2. **Cálculo do Novo ID**: Calcula o próximo ID disponível para o novo anúncio, buscando o maior ID existente e somando um.
3. **Inserção do Anúncio**: Insere um novo registro na tabela Announcement com os dados fornecidos e o novo ID calculado.

**Procedure: update\_announcement**

A procedure update\_announcement tem a função de atualizar um anúncio existente na tabela Announcement.

**Script:**

CREATE OR REPLACE PROCEDURE insert\_announcement(

p\_community\_id INT,

p\_sender\_id VARCHAR,

p\_sender\_name VARCHAR,

p\_image TEXT,

p\_description VARCHAR,

p\_tag VARCHAR

)

LANGUAGE plpgsql AS $$

DECLARE

v\_new\_id INT;

BEGIN

*-- Calcula o próximo ID*

SELECT COALESCE(MAX(ID), 0) + 1 INTO v\_new\_id FROM Announcement;

INSERT INTO Announcement (ID, community\_id, sender\_id, sender\_name, image, description, tag)

VALUES (v\_new\_id, p\_community\_id, p\_sender\_id, p\_sender\_name, p\_image, p\_description, p\_tag);

END;

$$;

1. **Parâmetro de Entrada**: Recebe o ID do anúncio que deve ser atualizado, junto com os novos valores para a comunidade, remetente, imagem, descrição e tag.
2. **Atualização do Anúncio**: Executa uma instrução UPDATE na tabela Announcement, alterando os valores dos campos correspondentes ao ID fornecido.

**Procedure: delete\_announcement**

A procedure delete\_announcement tem a finalidade de remover um anúncio da tabela Announcement.

**Script:**

CREATE OR REPLACE PROCEDURE delete\_announcement(

p\_announcement\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

DELETE FROM Announcement WHERE ID = p\_announcement\_id;

END;

$$;

1. **Parâmetro de Entrada**: Recebe o ID do anúncio que deve ser excluído.
2. **Exclusão do Anúncio**: Executa uma instrução DELETE na tabela Announcement, removendo o registro que corresponde ao ID fornecido.

Procedures da tabela publication:

**Procedure:** insert\_publication  
A procedure insert\_publication é responsável por inserir uma nova publicação na tabela Publication.

**Script:**

CREATE OR REPLACE PROCEDURE insert\_publication(

p\_date DATE,

p\_likes INT,

p\_description VARCHAR,

p\_customer\_cpf VARCHAR

)

LANGUAGE plpgsql AS $$

DECLARE

v\_customer\_id INT;

v\_new\_id INT;

BEGIN

SELECT ID INTO v\_customer\_id FROM Customer WHERE cpf = p\_customer\_cpf;

IF NOT FOUND THEN

RAISE EXCEPTION 'Customer not found with CPF: %', p\_customer\_cpf;

END IF;

*-- Calcula o próximo ID*

SELECT COALESCE(MAX(ID), 0) + 1 INTO v\_new\_id FROM Publication;

INSERT INTO Publication (ID, publication\_date, likes, description, customer\_id)

VALUES (v\_new\_id, p\_date, p\_likes, p\_description, v\_customer\_id);

END;

$$;

1. **Parâmetros de Entrada:** Recebe os dados necessários para criar uma publicação, incluindo a data, número de likes, descrição e CPF do cliente.
2. **Validação do Cliente:** Busca o ID do cliente na tabela Customer usando o CPF fornecido. Se o cliente não for encontrado, gera uma exceção.
3. **Cálculo do Novo ID:** Calcula o próximo ID disponível para a nova publicação, utilizando o maior ID atual da tabela Publication.
4. **Inserção da Publicação:** Executa uma instrução INSERT para adicionar a nova publicação na tabela Publication, preenchendo os campos com os dados fornecidos e o ID obtido na etapa anterior.

**Procedure:** update\_publication  
A procedure update\_publication é responsável por atualizar os dados de uma publicação existente na tabela Publication.

**Script:**

CREATE OR REPLACE PROCEDURE update\_publication(

p\_field VARCHAR,

p\_new\_value VARCHAR,

p\_publication\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

EXECUTE format('UPDATE Publication SET %I = $1 WHERE ID = $2', p\_field)

USING p\_new\_value, p\_publication\_id;

END;

$$;

1. **Parâmetros de Entrada:** Recebe os dados necessários para atualizar uma publicação, incluindo o ID da publicação, a nova data, o número de likes e a descrição.
2. **Atualização da Publicação:** Executa uma instrução UPDATE para modificar os campos publication\_date, likes e description da publicação na tabela Publication, onde o ID corresponde ao ID fornecido.

**Procedure:** delete\_publication  
A procedure delete\_publication é responsável por excluir uma publicação existente da tabela Publication.

**Script:**

CREATE OR REPLACE PROCEDURE delete\_publication(

p\_publication\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

DELETE FROM Publication WHERE ID = p\_publication\_id;

END;

$$;

1. **Parâmetros de Entrada:** Recebe o ID da publicação a ser excluída.
2. **Exclusão da Publicação:** Executa uma instrução DELETE para remover a publicação da tabela Publication, onde o ID corresponde ao ID fornecido.

Procedures da tabela community

**Procedure:** insert\_community

A procedure insert\_community é responsável por inserir uma nova comunidade na tabela Community e adicionar um administrador a ela.

**Script:**

CREATE OR REPLACE PROCEDURE insert\_community(

p\_name VARCHAR,

p\_date DATE,

p\_image TEXT,

p\_customer\_cpf VARCHAR

)

LANGUAGE plpgsql AS $$

DECLARE

v\_community\_id INT;

v\_new\_id INT;

BEGIN

*-- Calcula o próximo ID*

SELECT COALESCE(MAX(ID), 0) + 1 INTO v\_new\_id FROM Community;

INSERT INTO Community (ID, name, start\_date, image)

VALUES (v\_new\_id, p\_name, p\_date, p\_image)

RETURNING ID INTO v\_community\_id;

CALL add\_administrator(p\_customer\_cpf, v\_community\_id);

END;

$$;

1. **Parâmetros de Entrada:** Recebe os dados necessários para criar uma comunidade, incluindo o nome, a data de início, a imagem e o CPF do cliente que será o administrador.
2. **Cálculo do Novo ID:** Calcula o próximo ID disponível para a nova comunidade, utilizando o maior ID atual da tabela Community.
3. **Inserção da Comunidade:** Executa uma instrução INSERT para adicionar a nova comunidade na tabela Community, preenchendo os campos com os dados fornecidos e retornando o ID da nova comunidade.
4. **Adição do Administrador:** Chama a procedure add\_administrator para associar o cliente com o CPF fornecido como administrador da nova comunidade.

**Procedure:** update\_community

A procedure update\_community é responsável por atualizar os dados de uma comunidade existente na tabela Community.

**Script:**

CREATE OR REPLACE PROCEDURE update\_community(

p\_field VARCHAR,

p\_new\_value VARCHAR,

p\_community\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

EXECUTE format('UPDATE Community SET %I = $1 WHERE ID = $2', p\_field)

USING p\_new\_value, p\_community\_id;

END;

$$;

1. **Parâmetros de Entrada:** Recebe os dados necessários para atualizar uma comunidade, incluindo o ID da comunidade, o novo nome, a nova data de início e a nova imagem.
2. **Atualização da Comunidade:** Executa uma instrução UPDATE para modificar os campos name, start\_date e image da comunidade na tabela Community, onde o ID corresponde ao ID fornecido.

**Procedure:** delete\_community

A procedure delete\_community é responsável por excluir uma comunidade existente da tabela Community.

**Script:**

CREATE OR REPLACE PROCEDURE delete\_community(

p\_community\_id INT

)

LANGUAGE plpgsql AS $$

BEGIN

DELETE FROM Community WHERE ID = p\_community\_id;

END;

$$;

1. **Parâmetros de Entrada:** Recebe o ID da comunidade a ser excluída.
2. **Exclusão da Comunidade:** Executa uma instrução DELETE para remover a comunidade da tabela Community, onde o ID corresponde ao ID fornecido.

Outras procedures:

**Procedure:** add\_customer\_to\_community

A procedure add\_customer\_to\_community é responsável por adicionar um cliente a uma comunidade existente na tabela Community\_Customer.

**Script:**

CREATE OR REPLACE PROCEDURE add\_customer\_to\_community(

p\_cpf VARCHAR,

p\_community\_id INT

)

LANGUAGE plpgsql AS $$

DECLARE

v\_customer\_id INT;

BEGIN

SELECT ID INTO v\_customer\_id FROM Customer WHERE cpf = p\_cpf;

IF NOT FOUND THEN

RAISE EXCEPTION 'Customer not found with CPF: %', p\_cpf;

END IF;

INSERT INTO Community\_Customer (customer\_id, community\_id)

VALUES (v\_customer\_id, p\_community\_id);

END;

$$;

1. **Parâmetros de Entrada:** Recebe o CPF do cliente e o ID da comunidade à qual o cliente será adicionado.
2. **Validação do Cliente:** Busca o ID do cliente na tabela Customer usando o CPF fornecido. Se o cliente não for encontrado, gera uma exceção.
3. **Inserção na Tabela de Comunidade:** Executa uma instrução INSERT para adicionar o cliente à tabela Community\_Customer, associando o ID do cliente ao ID da comunidade fornecido.

**Procedure:** add\_administrator

A procedure add\_administrator é responsável por adicionar um cliente como administrador de uma comunidade na tabela Administrator.

**Script:**

CREATE OR REPLACE PROCEDURE add\_administrator(

p\_customer\_cpf VARCHAR,

p\_community\_id INT

)

LANGUAGE plpgsql AS $$

DECLARE

v\_customer\_id INT;

BEGIN

*-- Obter o ID do cliente com base no CPF fornecido*

SELECT ID INTO v\_customer\_id FROM Customer WHERE cpf = p\_customer\_cpf;

IF NOT FOUND THEN

RAISE EXCEPTION 'Customer not found with CPF: %', p\_customer\_cpf;

END IF;

*-- Inserir o ID do cliente na tabela Administrator*

INSERT INTO Administrator (customer\_id, community\_id)

VALUES (v\_customer\_id, p\_community\_id);

END; $$;

1. **Parâmetros de Entrada:** Recebe o CPF do cliente e o ID da comunidade na qual o cliente será adicionado como administrador.
2. **Validação do Cliente:** Busca o ID do cliente na tabela Customer usando o CPF fornecido. Se o cliente não for encontrado, gera uma exceção.
3. **Inserção na Tabela de Administrador:** Executa uma instrução INSERT para adicionar o ID do cliente à tabela Administrator, associando-o à comunidade especificada.

**Function:**

**O que é uma function?**

Uma *function* (ou função) em SQL é um bloco de código que executa uma tarefa específica e retorna um valor. Ao contrário das *procedures*, que executam ações, mas não precisam retornar valores, as *functions* sempre produzem um resultado. Elas são úteis para cálculos, validações, ou qualquer operação que precise gerar uma resposta — como verificar se um usuário é administrador de uma comunidade, por exemplo. Functions podem ser chamadas em consultas SQL, o que facilita reutilizar lógica e evitar repetição de código no banco de dados.

**Functions no banco de dados do segundo ano:**

**Function:** check\_administrator

A função check\_administrator verifica se um cliente específico é administrador de uma determinada comunidade.

**Script:**

CREATE OR REPLACE FUNCTION check\_administrator(

p\_customer\_cpf VARCHAR,

p\_community\_id INT

) RETURNS BOOLEAN AS $$

DECLARE

v\_exists BOOLEAN;

BEGIN

SELECT EXISTS (

SELECT 1 FROM Administrator

WHERE customer\_cpf = p\_customer\_cpf AND community\_id = p\_community\_id

) INTO v\_exists;

RETURN v\_exists;

END;

$$ LANGUAGE plpgsql;

1. **Parâmetros de Entrada:** Recebe o CPF do cliente e o ID da comunidade para verificar a associação.
2. **Verificação de Existência:** Executa uma consulta para verificar se existe um registro na tabela Administrator com o CPF do cliente e o ID da comunidade fornecidos, armazenando o resultado na variável v\_exists.
3. **Retorno:** A função retorna um valor booleano (TRUE ou FALSE), indicando se o cliente é ou não administrador da comunidade.

**Triggers:**

**O que são triggers?**

**Triggers** são objetos do banco de dados que executam automaticamente uma função quando uma ação específica (como INSERT, UPDATE ou DELETE) ocorre em uma tabela. Eles são usados para automatizar auditorias e manter a consistência dos dados, permitindo que o banco registre ações sem intervenção manual.

**Triggers no banco de dados do segundo ano:**

CREATE OR REPLACE FUNCTION log\_customer\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Customer (action\_type, customer\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_customer

AFTER INSERT OR UPDATE OR DELETE ON Customer

FOR EACH ROW EXECUTE FUNCTION log\_customer\_action();

CREATE OR REPLACE FUNCTION log\_advertisement\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Advertisement (action\_type, advertisement\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_advertisement

AFTER INSERT OR UPDATE OR DELETE ON Advertisement

FOR EACH ROW EXECUTE FUNCTION log\_advertisement\_action();

CREATE OR REPLACE FUNCTION log\_announcement\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Announcement (action\_type, announcement\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_announcement

AFTER INSERT OR UPDATE OR DELETE ON Announcement

FOR EACH ROW EXECUTE FUNCTION log\_announcement\_action();

CREATE OR REPLACE FUNCTION log\_publication\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Publication (action\_type, publication\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_publication

AFTER INSERT OR UPDATE OR DELETE ON Publication

FOR EACH ROW EXECUTE FUNCTION log\_publication\_action();

CREATE OR REPLACE FUNCTION log\_community\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Community (action\_type, community\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_community

AFTER INSERT OR UPDATE OR DELETE ON Community

FOR EACH ROW EXECUTE FUNCTION log\_community\_action();

CREATE OR REPLACE FUNCTION log\_plan\_action() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO Log\_Plan (action\_type, plan\_id)

VALUES (TG\_OP, NEW.ID);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger\_log\_plan

AFTER INSERT OR UPDATE OR DELETE ON Plan

FOR EACH ROW EXECUTE FUNCTION log\_plan\_action();

**Funções de Log:**

Cada função de log insere uma entrada na tabela de log correspondente (Log\_Customer, Log\_Advertisement, etc.) com o tipo de ação (TG\_OP) e o ID do registro afetado (NEW.ID).

**Funções das triggers**

Para cada tabela (como Customer, Advertisement, Announcement, etc.), um *trigger* é configurado para acionar a função de log correspondente após cada inserção, atualização ou exclusão, registrando automaticamente as alterações realizadas na tabela específica.

Esse sistema de *triggers* e funções ajuda a manter um histórico de ações, facilitando o monitoramento e a auditoria das alterações no banco de dados.

**Conclusão geral**