IMPLEMENTAÇÃO DE UM CRUD INTEGRADO: PostgreSQL com Funções de Banco de Dados e Aplicação em C#/.NET

João Gabriel Manhoni
Graduando em Banco de dados pela Fatec Bauru
joao.manhoni@fatec.sp.gov.br

Luis Alexandre da Silva
Mestre em Ciência da Computação e Docente na Fatec Bauru
luis.silva51@fatec.sp.gov.br

RESUMO

Este trabalho apresenta a implementação de um sistema CRUD integrado entre PostgreSQL e C#/.NET, visando centralizar a lógica de negócio no banco de dados por meio de *functions* e *triggers* em PL/pgSQL. A proposta busca reforçar a integridade e a segurança de sistemas transacionais, reduzindo vulnerabilidades comuns, como SQL *Injection*, e evitando duplicação de regras entre aplicação e banco. A metodologia envolveu a modelagem de tabelas relacionais, criação de *functions* de validações diretamente no banco e desenvolvimento de uma aplicação C#/WPF estruturada em MVVM, com comunicação ao banco via Npgsql. Os resultados evidenciaram que a abordagem adotada garante consistência, modularidade e robustez, confirmando a viabilidade da integração como referência para projetos acadêmicos e profissionais.

Palavras-chave: PostgreSQL; C#; CRUD; MVVM; Segurança.

1 INTRODUÇÃO

O avanço das tecnologias de informação exige boas práticas no desenvolvimento de aplicações com bancos de dados, onde a integridade é fundamental para a qualidade do sistema (Date, 2003). Desde o modelo relacional de Codd (1970), a complexidade no armazenamento demanda soluções que garantam as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID) para transações confiáveis e sejam seguras contra vulnerabilidades como SQL *Injection* (Castelano, [s.d.]; Silberschatz *et al*., 2020).

Uma problemática comum é a duplicação de regras de negócio entre aplicação e banco, mitigável com encapsulação da lógica em *stored procedures* (Elmasri; Navathe, 2016). Arquiteturas mal definidas comprometem integridade e escalabilidade em ambientes de alta concorrência (Codd, 1970). Este trabalho propõe um sistema que centraliza a lógica de negócio no PostgreSQL, reconhecido por sua conformidade ACID (Momjian, 2001), integrado a uma interface C# com Windows Presentation Foundation (WPF).

A justificativa reside na necessidade de sistemas alinhados aos padrões corporativos, onde a robustez arquitetural é crítica (Esposito, 2018). A solução combina PostgreSQL, que oferece segurança e extensibilidade (Momjian, 2001), com o padrão *Model-View-ViewModel* (MVVM) que consiste na separação clara entre dados (*Model*), lógica de controle (*ViewModel*) e apresentação (*View*), garantindo manutenibilidade através da separação entre interface e lógica de negócio (Richter, 2022).

Este projeto tem como objetivo demonstrar a construção de um sistema CRUD (*Create, Read, Update, Delete*) diretamente integrado ao banco de dados PostgreSQL, utilizando funções armazenadas (*stored functions*) para realizar operações e validações. Ao centralizar a lógica de negócio no próprio banco, o sistema garante maior consistência dos dados (Momjian, 2001), respeitando as propriedades ACID, fundamentais para a integridade das transações(Silberschatz *et al.*, 2020). Como prova de conceito, é desenvolvida uma aplicação em C# com WPF utilizando o padrão arquitetural MVVM, que segundo Richter (2022), permite validar na prática a integração entre a camada de apresentação e o banco de dados, demonstrando como a lógica centralizada pode ser acessada e utilizada de forma eficiente por aplicações reais. Essa abordagem facilita a visualização do funcionamento do sistema e reforça a aplicabilidade da solução em cenários concretos.

2 REFERENCIAL TEÓRICO

2.1 TECNOLOGIAS: Banco de Dados Relacional e SQL

Bancos de dados (BD) são sistemas fundamentais para o armazenamento estruturado de informações persistentes (Date, 2003). Stonebraker (2010) destacou a evolução para modelos como o relacional, baseado em tabelas com chaves primárias (PK - *Primary Key*) e estrangeiras (FK - *Foreign Key*), além dos NoSQL, voltados a cenários de *Big Data*.

O modelo relacional de Codd (1970) organiza dados em tabelas, garantindo integridade por meio de PKs, FKs e operações fundamentadas em álgebra relacional (Silberschatz; Korth; Sudarshan, 2020). Chen (1976) complementou esse modelo com o Entidade-Relacionamento (ER), que representa dados conceituais por entidades e relacionamentos.

A normalização elimina redundâncias e anomalias (Elmasri; Navathe, 2016), enquanto a linguagem SQL, padronizada pelo American National Standards Institute (ANSI), oferece sintaxe acessível para consultas declarativas (Melton; Simon, 2002), como no exemplo: “SELECT * FROM Clientes WHERE cidade = 'Bauru';”.

O padrão ANSI não apenas define a estrutura e semântica da linguagem SQL, mas também especifica códigos padronizados para tratamento de erros e exceções, conhecidos como SQLSTATE. Esses códigos, compostos por cinco caracteres alfanuméricos, permitem que aplicações identifiquem e tratem falhas de forma consistente e independente do SGBD utilizado, fortalecendo a interoperabilidade e a confiabilidade dos sistemas (Melton; Simon, 2002).

Entre os sistemas gerenciadores de Banco de Dados (SGBDs), o PostgreSQL destaca-se por ser *open-source*, robusto, extensível e aderente aos princípios ACID. Sua arquitetura avançada inclui suporte a tipos complexos como JSON, *arrays* e estruturas compostas, sua extensibilidade permite a criação de tipos personalizados, operadores e funções específicas, tornando-o uma solução versátil (Momjian, 2001). O PostgreSQL ainda oferece mecanismos de segurança e gerenciamento por *schemas* (Postgresql Global Development Group, 2024), embora este trabalho foque especificamente em validação por *functions* e *triggers*.

O Quadro 1 resume diferenças-chave entre PostgreSQL e outros SGBDs populares, onde se destaca como principal diferença a conformidade ACID completa, sem deixar de esquecer o fato de que é um software *Open-source* e ainda possui uma linguagem robusta como o PL/pgSQL.

Quadro 1 – Comparação entre PostgreSQL, MySQL e SQL Server

Recurso	PostgreSQL	MySQL	SQL Server
Conformidade ACID	Completa	Parcial	Completa
Extensibilidade	PL/pgSQL	Stored procedures básicas	T-SQL avançado
Licença	Open-source (BSD)	Open-source (GPL)	Proprietário

Fonte: Adaptado de Silberschatz *et al.*, 2020 e Dias; O, 2023.

Nesse contexto, compreender e aplicar operações de manipulação de dados como inserção, leitura, atualização e exclusão é fundamental para a prática em SGBDs. Essas operações, conhecidas como CRUD, são exploradas na seção seguinte.

2.2 TECNOLOGIA: CRUD (*Create, Read, Update, Delete*)

As operações conhecidas como CRUD, *Create* (criar), *Read* (ler), *Update* (atualizar) e *Delete* (excluir), representam a base de qualquer sistema de banco de dados. Segundo Silberschatz; Korth; Sudarshan (2020), constituem os pilares do modelo relacional, garantindo consistência e controle na manipulação dos dados.

No PostgreSQL, o CRUD é implementado por meio da linguagem SQL, utilizando comandos como INSERT, SELECT, UPDATE e DELETE. A padronização do SQL pela ANSI proporcionou uma sintaxe unificada que facilita a interoperabilidade entre diferentes SGBDs (Elmasri; Navathe, 2016). Além disso, essas operações podem ser encapsuladas em *functions* e *triggers* escritas em PL/pgSQL, agregando validações adicionais e fortalecendo a integridade transacional (Momjian, 2001).

A Figura 1 ilustra um exemplo de função PL/pgSQL para validação de dados, onde o código faz uma validação de CPF simples, exigindo que seja inserido uma *string* e verificando se ele tem 11 caracteres, retornando um boleano (true/false). Isso demonstra como operações CRUD podem ser expandidas como regras de negócio.

Figura 1 - Um exemplo prático de função PL/pgSQL para validação de dados.

```
1      CREATE OR REPLACE FUNCTION validar_cliente(cpf VARCHAR) RETURNS
              BOOLEAN AS $$
2      BEGIN
3             IF cpf IS NULL OR cpf !~ '^[0-9]{11}$' THEN
4                    RETURN FALSE;
5             END IF;
6                    RETURN TRUE;
7      END;
8      $$ LANGUAGE plpgsql;
```

Adaptado de Elmasri; Navathe, 2016.

Além das funções, um aspecto essencial na implementação segura e robusta de operações CRUD é o uso dos códigos SQLSTATE, definidos pelo padrão ANSI. Esses códigos, compostos por cinco caracteres alfanuméricos, representam o estado de execução de uma operação e permitem identificar precisamente o tipo de erro ocorrido, como violação de chave primária, falha de integridade referencial ou erro de sintaxe. No contexto de aplicações conectadas a bancos de dados, o uso do SQLSTATE viabiliza o tratamento padronizado de exceções na camada de aplicação, permitindo

respostas específicas para cada tipo de falha e facilitando a manutenção e a depuração do sistema (Melton; Simon, 2002).

Outro fator são as *constraints* como PRIMARY KEY, FOREIGN KEY e CHECK asseguram integridade referencial, prevenindo falhas como duplicidade ou remoção de registros ainda referenciados (Dias; Oliveira, 2023). Dessa forma, o CRUD não deve ser entendido apenas como comandos básicos, mas como uma abordagem arquitetural que organiza a interação entre aplicações e bancos de dados, apoiando-se em boas práticas de segurança e eficiência (Silberschatz *et al.*, 2020; Elmasri; Navathe, 2016).

A implementação correta dessas operações no PostgreSQL garante persistência e integridade do sistema. Contudo, para que sejam acessíveis ao usuário final, é necessário integrá-las a uma aplicação com interface gráfica, papel desempenhado pela linguagem C#/WPF, discutida na próxima seção.

2.3 Linguagem de Programação C#/WPF

A linguagem C#, desenvolvida pela Microsoft, destaca-se por sua orientação a objetos, segurança de tipagem e integração em aplicações corporativas. Combinada ao WPF (Windows Presentation Foundation), oferece recursos avançados para o desenvolvimento de aplicações *desktop* modernas e responsivas (Microsoft, 2024a; Esposito, 2018).

O WPF adota XAML (*Extensible Application Markup Language*), separando a camada visual (*View*) da lógica de negócio (*ViewModel*), em conformidade com o padrão MVVM. Esse padrão separa claramente as responsabilidades entre lógica de negócio e interface, promovendo coesão, reutilização e testabilidade (Richter, 2022). Recursos como Data Binding, Observable Collections e Dependency Properties permitem que alterações nos dados sejam refletidas automaticamente na interface (Microsoft, 2024b). Complementarmente, conceitos como Dependency Injection e IoC containers aumentam modularidade e reduzem acoplamento (Fowler, 2004; Microsoft, 2024d).

A integração entre C# e PostgreSQL é viabilizada pela biblioteca Npgsql, que permite executar comandos SQL, chamadas a *stored procedures*, *triggers* e *functions*, centralizando operações de negócio de forma segura (Momjian, 2001). Além disso, a linguagem oferece suporte a manipulação assíncrona e controle de eventos, possibilitando interfaces responsivas que reagem a mudanças em tempo real (Wiener, 2004).

No aspecto da segurança, Gonçalves Junior (2013) alerta para falhas como SQL *Injection*, que decorrem de entradas não validadas e comandos dinâmicos inseguros. Práticas como *prepared statements*, validação no cliente e centralização de regras no banco via *triggers*, *functions* e *constraints* são essenciais para mitigá-las. Vissotto Junior. *et al.* (2024) reforçam que a combinação entre validações no banco e boas práticas de código resulta em sistemas mais robustos e resilientes.

Assim, a utilização de C#/WPF aliado ao PostgreSQL compõe uma plataforma robusta para sistemas corporativos. A aplicação do padrão MVVM, somada às boas práticas de integração e segurança, garante aplicações críticas com alta coesão, baixo acoplamento e resistência a vulnerabilidades.

3 MATERIAIS E MÉTODOS

Este trabalho adotou uma abordagem aplicada e exploratória, com o objetivo de demonstrar a integração prática entre um sistema de banco de dados relacional (PostgreSQL) e uma aplicação desenvolvida em C# utilizando o *framework* WPF. A proposta apresenta a construção de um sistema CRUD com foco na centralização da lógica de negócio por meio de funções armazenadas no banco de dados, reforçando conceitos de segurança, modularidade e desempenho.

A metodologia foi dividida em três etapas principais:

a) definição da estrutura de banco de dados;
b) implementação das funções em PL/pgSQL;
c) desenvolvimento da interface gráfica com integração ao PostgreSQL.

A estrutura lógica do banco foi modelada utilizando o modelo relacional, com definição de PKs e FKs, *constraints* e validações diretamente no SGBD.

Para a camada de aplicação, é utilizada a linguagem C# com a biblioteca Npgsql para comunicação com o banco. A arquitetura adotada na aplicação foi baseada no padrão MVVM através do WPF, com uso de Data Binding, Commands, Dependency Properties e Dependency Injection, conforme as boas práticas da Microsoft (2024c; 2024d) e Fowler (2004). A separação entre a lógica de apresentação e a lógica de acesso a dados foi projetada para manter um código modular, reutilizável e fácil de manter, além de garantir uma interface dinâmica e responsiva.

Para o desenvolvimento, foi utilizado um ambiente local de testes com recursos amplamente empregados no contexto profissional. O sistema operacional adotado é o Windows 11, sobre o qual foi executado o Visual Studio 2022 com suporte à plataforma .NET 8.0. A linguagem de programação utilizada é o C#, associada ao *framework* .NET Core 8 com um projeto WPF, que permite a construção de interfaces gráficas ricas e desacopladas da lógica de controle.

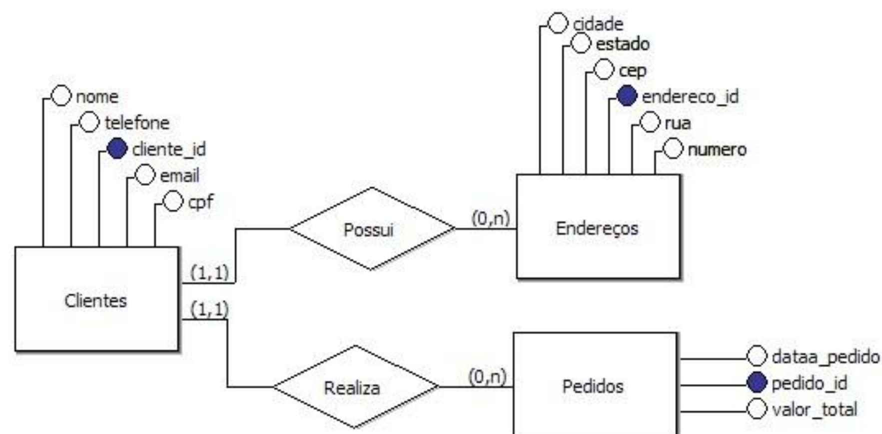
Como sistema gerenciador de banco de dados, foi utilizado o PostgreSQL em sua versão 17.2, administrado por meio da ferramenta gráfica pgAdmin 4, versão 9.0. A comunicação entre a aplicação e o banco de dados aconteceu por meio da biblioteca Npgsql, que oferece suporte à execução de comandos SQL e à chamada de funções e procedimentos armazenados.

A arquitetura escolhida para o desenvolvimento segue o padrão MVVM, que favorece a separação de responsabilidades e a organização do código, permitindo a reutilização de componentes e facilitando futuras manutenções. Além disso, foi aplicada a técnica de injeção de dependência, que contribui para a modularização e a testabilidade da aplicação.

A implementação seguiu um fluxo dividido em três fases principais. Inicialmente, foi feita a modelagem das tabelas de dados com base no modelo relacional, incluindo definição de chaves primárias e estrangeiras, uso de *constraints* como *NOT NULL, DEFAULT, UNIQUE*, além da criação de *stored procedures e functions* escritas em PL/pgSQL. Essa abordagem visa encapsular a lógica de negócio no banco, minimizando erros e vulnerabilidades. As tabelas do banco de dados foram criadas seguindo o modelo relacional, com definição de chaves primárias e estrangeiras, além de restrições de integridade. A Figura 2 apresenta o modelo conceitual do banco de dados, ilustrando as entidades, atributos e relacionamentos que estruturam o sistema. As entidades *Clientes*, *Endereços* e *Pedidos* são representadas por retângulos, enquanto os relacionamentos são mostrados por losangos, que indicam como essas entidades se conectam. As cardinalidades entre parênteses mostram que cada cliente

pode possuir vários endereços e realizar diversos pedidos, mas cada endereço e pedido estão sempre associados a um único cliente. A tabela Clientes contém os campos cliente_id (PK SERIAL), nome (VARCHAR(100)), cpf (VARCHAR(11), com restrição UNIQUE), email (VARCHAR(100)) e telefone (VARCHAR(15)). A tabela Endereços possui endereco_id (PK SERIAL), cliente_id (FK), rua (VARCHAR(150)), numero (VARCHAR(10)), cidade (VARCHAR(100)), estado (VARCHAR(2)) e cep (VARCHAR(9)). Já Pedidos contém pedido_id (PK SERIAL), cliente_id (FK), data_pedido (DATE com valor padrão CURRENT_DATE) e valor_total (DECIMAL(10,2)). Essa estrutura garante integridade referencial entre as tabelas, assegurando que apenas clientes válidos possam ter endereços e pedidos associados, conforme demonstrado no diagrama.

Figura 2 – Esquema Conceitual do Banco de Dados.



Fonte: O Autor, 2025.

A Figura 3 apresenta o *script* utilizado para a criação das tabelas no PostgreSQL. O código SQL a seguir representa exatamente o esquema conceitual exibido acima. Repare que na tabela Clientes (linha 2) foram utilizadas *constraints* como PRIMARY KEY(3) no atributo ‘cliente_id’ e NOT NULL(4) no ‘nome’, pois o ‘cliente_id’ é o atributo chave desta tabela e o cliente não pode ser cadastrado sem nome.

Além da estrutura, foram implementadas *functions* e *triggers* de validação em PL/pgSQL, assegurando que operações de manipulação de dados seguissem regras pré-definidas. A Figura 4 apresenta a *function* de INSERT 'inserir_cliente', que recebe como parâmetros de entrada os dados do cliente (p_nome, p_cpf, p_email, p_telefone) e retorna um código inteiro indicando o resultado da operação. A *function* implementa um fluxo de validações sequenciais antes da inserção, como verificação de CPF (linhas 10 e 14) e se nome é nulo (linha 6). Apenas após todas as validações serem aprovadas, a *function* executa o INSERT na tabela Clientes e retorna o código 0 indicando sucesso na linha 20. O bloco EXCEPTION captura erros inesperados retornando o SQLState (linha 24) como código no padrão ANSI.

Figura 3 – SQL de criação das tabelas no banco.

```
1      -- Tabela Clientes
2      CREATE TABLE Clientes (
3          cliente_id SERIAL PRIMARY KEY,
4          nome VARCHAR(100) NOT NULL,
5          cpf VARCHAR(11) UNIQUE NOT NULL,
6          email VARCHAR(100),
7          telefone VARCHAR(15)
8      );
9
10     -- Tabela Enderecos
11     CREATE TABLE Enderecos (
12         endereco_id SERIAL PRIMARY KEY,
13         cliente_id INT NOT NULL,
14         rua VARCHAR(150) NOT NULL,
15         numero VARCHAR(10),
16         cidade VARCHAR(100) NOT NULL,
17         estado VARCHAR(2) NOT NULL,
18         cep VARCHAR(9),
19         FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
20     );
21
22     -- Tabela Pedidos
23     CREATE TABLE Pedidos (
24         pedido_id SERIAL PRIMARY KEY,
25         cliente_id INT NOT NULL,
26         data_pedido DATE NOT NULL DEFAULT CURRENT_DATE,
27         valor_total DECIMAL(10,2) NOT NULL,
28         FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
29     );
```

Fonte: O Autor, 2025.

As *functions* de UPDATE e SELECT, assim como funções do C# que tratam os retornos destas *functions*, se encontram em um repositório no GitHub[1], em que consta um Readme.md explicando o seu conteúdo.

Figura 4 – *Function* de Insert com validações.

```
1    CREATE OR REPLACE FUNCTION inse-
     rir_cliente(p_nome VARCHAR, p_cpf
     VARCHAR, p_email VARCHAR, p_telefone
     VARCHAR ) RETURNS INT AS $$
2    DECLARE
3       v_qtde INT; v_state TEXT;
4    BEGIN
5    -- Validação: nome não pode ser nulo
6    IF p_nome IS NULL OR LENGTH
     (TRIM(p_nome)) = 0 THEN
7       RETURN -100; -- Nome inválido
8    END IF;
9    -- Validação: CPF deve ter 11 dígitos
      e não pode repetir
10   IF LENGTH(p_cpf) <> 11 OR p_cpf !~
     '^[0-9]+$' THEN
11      RETURN -101; -- CPF inválido
12   END IF;
13   SELECT COUNT(*) INTO v_qtde FROM Cli-
     entes WHERE cpf = p_cpf;
14   IF v_qtde > 0 THEN
15       RETURN -102; -- CPF já existe
16   END IF;
17   -- Se passou nas validações, insere
18   INSERT INTO Clientes (nome, cpf,
     email, telefone)
19   VALUES (p_nome, p_cpf, p_email,
     p_telefone);
20       RETURN 0; -- Sucesso
21   EXCEPTION
22       WHEN others THEN
23           GET STACKED DIAGNOSTICS
     v_state = RETURNED_SQLSTATE;
24           RETURN v_state;
25   END;
26   $$ LANGUAGE plpgsql;
```

Fonte: O Autor, 2025.

[1] Disponível em: https://github.com/Manhoni/TGCRUD. Acesso em: 19 set. 2025.

Na Figura 5 é apresentada uma *trigger* de validação aplicada à operação DELETE, cujo objetivo é impedir a exclusão de clientes que possuam pedidos associados. A função validar_delete_cliente() é executada automaticamente antes da remoção de um registro na tabela Clientes. Na linha 6, é realizada uma consulta à tabela Pedidos, armazenando em v_qtde a quantidade de pedidos vinculados ao cliente_id que está sendo excluído (referenciado pela palavra-chave OLD). Em seguida, na linha 9, ocorre uma verificação condicional: se v_qtde for maior que zero, significa que existem pedidos associados, e uma exceção é lançada com a mensagem "Não é permitido excluir cliente com pedidos associados.", interrompendo a execução do DELETE. Caso contrário, o fluxo segue normalmente e, na linha 14, a função retorna OLD, sinalizando que o registro pode ser excluído com segurança. Essa abordagem garante a integridade referencial entre as tabelas Clientes e Pedidos diretamente no nível do banco de dados, evitando a remoção acidental de dados essenciais e reduzindo a necessidade de validações adicionais na camada da aplicação C#.

Esses mecanismos complementaram as restrições de integridade, garantindo que operações inválidas fossem bloqueadas automaticamente pelo banco.

Figura 5 – *Trigger* de Delete com validações.

```
1    CREATE OR REPLACE FUNCTION vali-
     dar_delete_cliente()RETURNS TRIGGER
     AS $$
2    DECLARE
3        v_qtde INT;
4    BEGIN
5    -- Verifica se o cliente tem pedidos

6        SELECT COUNT(*) INTO v_qtde
7        FROM Pedidos
8        WHERE cliente_id = OLD.cli-
     ente_id;
9        IF v_qtde > 0 THEN
10           RAISE EXCEPTION 'Não é permi-
     tido excluir cliente com pedidos  as-
     sociados.';
11       END IF;
12   RETURN OLD; -- permite o delete se
     não tiver pedidos
13   END;
14   $$ LANGUAGE plpgsql;
15
16   CREATE TRIGGER trg_validar_de-
     lete_cliente
17   BEFORE DELETE ON Clientes
18   FOR EACH ROW
19   EXECUTE FUNCTION validar_de-
     lete_cliente();
```

Fonte: O Autor, 2025.

Na segunda parte, no desenvolvimento da aplicação, a interface construída utilizando o padrão MVVM. A comunicação com o banco feita por meio de comandos parametrizados via Npgsql, evitando o uso de SQL dinâmico e prevenindo falhas como SQL *Injection*.

O *ViewModel* utilizou inversão de controle e injeção de dependência para organizar os serviços e repositórios da aplicação, conforme as práticas recomendadas por Fowler (2004) e Microsoft (2024d). Eventos em C# foram utilizados para controlar interações da interface com os dados.

A Figura 6 apresenta a interface gráfica de cadastro de clientes, onde se observam os campos de Nome, CPF, *Email* e Telefone, que são preenchidos ao selecionar o registro no DataGrid na parte inferior da tela, além dos botões de inserção, atualização, deleção e consulta dos registros do banco. Os códigos das funções dos botões estarão no GitHub.

Figura 6 – Tela de cadastro de Clientes.

Fonte: O Autor, 2025.

A aplicação foi desenvolvida em C# utilizando o padrão arquitetural MVVM no *framework* WPF.

Para executar operações no banco, como inserção e atualização, foram criadas as funções 'ExecuteFunction' e 'ExecuteSelect' em C#, que executam uma *function* PL/pgSQL e capturam seu valor de retorno. Na Figura 7 temos a função que executa a *function* de INSERT ou UPDATE, que é obtida pelo parâmetro 'functionName' (linha 1). A função constrói a *query* dinamicamente e utiliza comandos parametrizados via NpgsqlCommand (linhas 5-8), onde os parâmetros são adicionados de forma segura para prevenir SQL *Injection*. O bloco *try-catch* (linhas 9-23) implementa tratamento de exceções: em caso de sucesso, ExecuteScalar() captura e converte o retorno para inteiro (linhas 11-12); em caso de erro, PostgresException captura o código SQLSTATE do PostgreSQL (linha 19), identificando tipos específicos de falhas como violação de UNIQUE (23505).

Figura 7 – Função em C# para tratamento das respostas do PostgreSQL.

```
1    public static int ExecuteFunction
       (string functionName, Dictionary <
        string, object> parametros)
2    {
3        using var conn = GetConnection();
4        conn.Open();
5        string sql = $"SELECT {function-
     Name}({string.Join(",   ",   parame-
     tros.Keys)})";
6                using var cmd = new Npgsql-
     Command(sql, conn);
7        foreach (var param in parametros)
8            cmd.Parameters.AddWithValue(
             param.Key, param.Value ?? DB
             Null.Value);
9         try
10          {
11            object result =
              cmd.ExecuteScalar();
12            return result != null &&
             int.TryParse(result.ToS-
         tring(),
              out int codigo)
13            ? codigo : 0;
14        }
15      catch (PostgresException ex)
16        }
17   // Aqui você tem acesso ao SQLSTATE
18       if (int.TryParse(ex.SqlState,
         out int sqlStateCode))
19          return sqlStateCode;
20       else
21          return -1;
22       }
23   }
```

Fonte: O Autor, 2025.

Após a chamada, o valor retornado pela variável 'result' deve ser tratado de acordo com as regras de negócio da aplicação. Na utilização da função de *insert* optou-se pelo uso de *switch*, onde a *function* PL/pgSQL 'inserir_cliente' é chamada via C# e o código de retorno é tratado para exibir mensagens adequadas ao usuário. No

caso do *trigger* de DELETE, teremos um tratamento de exceção padrão da linguagem C#.

Com isso temos a implementação necessária para o tratamento das informações fornecidas pelas *functions* do PL/pgSQL.

Durante a parte de validações, foram realizados testes funcionais e de integração. Cada operação (inserção, edição, exclusão e leitura) foi verificada localmente, em ambiente controlado, com foco em testes de consistência de dados e de segurança (tentativas de inserção e deleção inválida).

As *functions* parametrizadas retornaram códigos que auxiliaram na identificação de falhas, como CPFs duplicados ou valores inválidos, enquanto o *trigger* garantiu integridade ao bloquear exclusões incorretas diretamente no SGBD. Esse conjunto de técnicas aumentou a confiabilidade do sistema e reforçou a centralização da lógica no banco de dados.

Os resultados foram avaliados qualitativamente com base no correto funcionamento dos fluxos e na aderência à proposta de arquitetura segura, modular e reutilizável.

4 RESULTADOS E DISCUSSÃO

Esta Seção apresenta os resultados obtidos com a implementação prática do sistema proposto, discutindo as decisões de modelagem, os *scripts* utilizados no banco de dados PostgreSQL, as *functions* e *triggers* em PL/pgSQL e a interface construída em C#/WPF.

Era esperado que o sistema desenvolvido fosse capaz de realizar as operações de cadastro, consulta, edição e exclusão de dados de forma confiável, mantendo a integridade das informações por meio de validações estruturadas no próprio banco de dados.

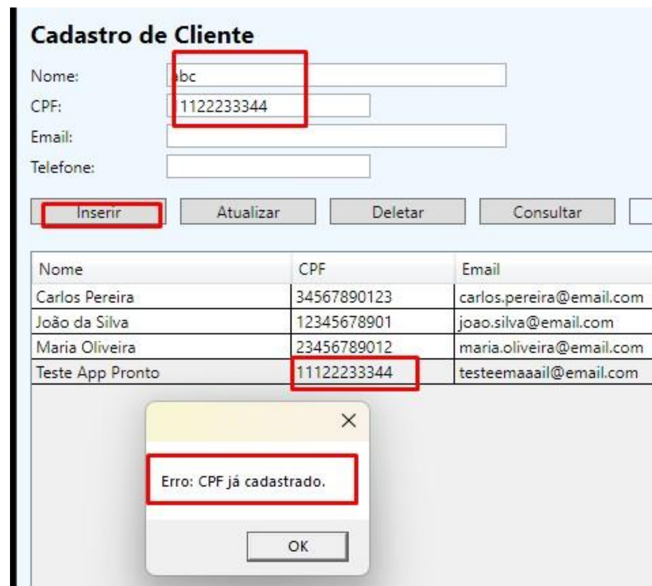
A criação das tabelas com *constraints* reforçou a integridade referencial (Codd, 1970; Silberschatz; Korth; Sudarshan, 2020). Esse procedimento garantiu que apenas dados válidos pudessem ser inseridos, evitando anomalias de atualização e exclusão. Durante os testes funcionais, operações de inserção, edição, exclusão e consulta mostraram-se consistentes, validando o alinhamento entre banco e aplicação.

A implementação prática do sistema atendeu ao objetivo central de demonstrar a viabilidade de um CRUD integrado entre PostgreSQL 17.2 e C#/WPF no .NET 8. A modelagem do banco de dados, fundamentada no modelo relacional proposto por Codd (1970), garantiu a integridade dos dados por meio de chaves primárias, estrangeiras e *constraints*. Assim, inserções inválidas, como CPFs duplicados, foram rejeitadas automaticamente (Figura 8), confirmando a eficácia das validações (Silberschatz; Korth; Sudarshan, 2020).

Além disso, o tratamento de exceções implementado nas *functions* PL/pgSQL, associado ao uso dos códigos SQLSTATE, possibilitou que o sistema identificasse erros específicos e retornasse mensagens adequadas diretamente à interface C#/WPF.

Essa integração entre o *backend* e a camada visual reforça a segurança contra falhas e melhora a experiência do usuário, que recebe *feedback* imediato sobre inconsistências, como mostrado na Figura 8, onde a tentativa de cadastrar um CPF já existente é corretamente bloqueada.

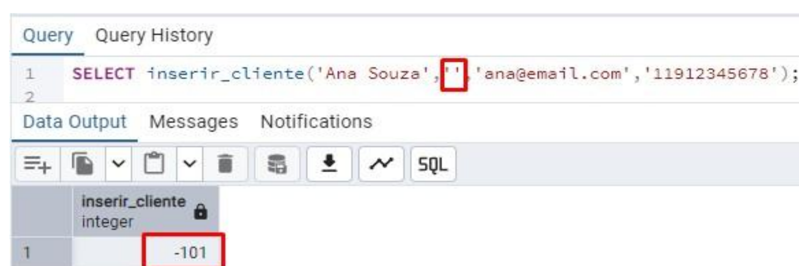
Figura 8 – Mensagem de erro ao tentar cadastrar cliente com CPF duplicado.



Fonte: O Autor, 2025.

As *functions* e *triggers* desenvolvidas em PL/pgSQL desempenharam papel central na centralização da lógica de negócio. Uma das *functions* de inserção, como mostrado na Figura 9, validava campos obrigatórios e retornava códigos de erro em caso de inconsistências, impedindo gravações incorretas. Esse tipo de abordagem contribuiu para a segurança e a consistência das operações, além de permitir a reutilização da lógica em diferentes partes do sistema. Conforme Gonçalves Junior (2013), a prevenção de SQL *Injection* pode ser fortalecida ao encapsular instruções SQL dentro de funções parametrizadas no SGBD.

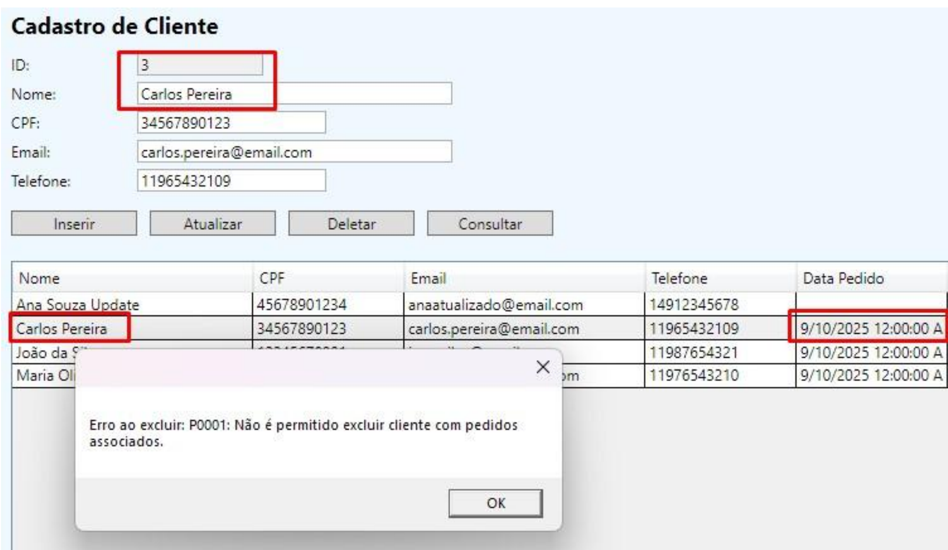
Figura 9 – Retorno da função PL/pgSQL ao tentar inserir cliente sem CPF.



Fonte: O Autor, 2025.

Já o *trigger* implementado bloqueou a exclusão de clientes com pedidos vinculados, mostrado na Figura 10, assegurando a integridade referencial entre as tabelas. Esse mecanismo confirmou, na prática, a importância da lógica encapsulada no banco. No âmbito da aplicação, a arquitetura baseada no padrão MVVM, aliada ao uso do WPF, proporcionou uma interface visual clara, responsiva e separada da lógica, facilitando futuras manutenções e evoluções do sistema.Esperava-se que a estrutura modular, com injeção de dependência e organização em camadas, contribuísse para a legibilidade e manutenibilidade do código-fonte, conforme as diretrizes da Microsoft (2024c; 2024d) e os princípios defendidos por Fowler (2004).

Figura 10 – Mensagem de erro ao tentar excluir cliente com pedidos associados.



Fonte: O Autor, 2025.

Segundo Esposito (2018), aplicações baseadas no MVVM tornam-se mais modulares e flexíveis, características confirmadas durante os testes do sistema.

O uso de comandos parametrizados via Npgsql eliminou a necessidade de SQL dinâmico, reforçando a segurança na comunicação com o banco (Esposito, 2018; Fowler, 2004; Microsoft, 2024c; 2024d). No caso do botão ‘Consultar’ não se faz necessária uma figura pois já é possível visualizar através das outras imagens a consulta sendo feita trazendo o pedido pela sua *function* que faz um *JOIN* no banco.

Em síntese, os resultados confirmaram que a integração entre PostgreSQL e C#/WPF, apoiada por validações no SGBD e por boas práticas arquiteturais no .NET, é capaz de produzir um sistema robusto, seguro e de fácil manutenção.

Esse resultado corroborou a discussão de Vissotto Junior. *et al.* (2024), que destacaram a importância de *prepared statements* para mitigar vulnerabilidades.

5 CONSIDERAÇÕES FINAIS

O projeto evidencia que medidas de segurança e integridade podem ser aplicadas de forma eficaz mesmo em ambientes locais e acadêmicos. A utilização de *constraints*, *functions* e *triggers* demonstra como a lógica de negócio pode ser centralizada no banco de dados, reduzindo inconsistências e reforçando a confiabilidade do sistema. Paralelamente, a aplicação em C#/WPF, estruturada em MVVM, comprova-se como solução viável para *front-ends* modulares e escaláveis.

Essas estratégias práticas, aplicáveis em ambientes domésticos ou de pequeno porte, demonstram que é possível aplicar medidas de segurança relevantes mesmo sem infraestrutura corporativa complexa, reforçando o papel do desenvolvedor na prevenção de falhas comuns desde as fases iniciais do projeto.

O sistema final não apenas cumpre os objetivos propostos, mas também pode servir como referência para futuros projetos acadêmicos e profissionais que demandem controle rigoroso sobre dados e validações em sistemas transacionais. Além disso, o modelo apresentado é flexível e pode ser exercitado em outros contextos tecnológicos, utilizando diferentes sistemas de gerenciamento de banco de dados re-

lacionais , como MySQL, SQL *Server* ou Oracle e diversas linguagens de programação, incluindo Java, Python ou PHP. Essa adaptabilidade permite explorar novos recursos que vêm sendo incorporados aos bancos de dados modernos, como *procedures* avançadas, *triggers* condicionais e suporte a tipos de dados complexos, mantendo-se alinhado às mesmas diretrizes de integridade, segurança e consistência transacional demonstradas neste trabalho. Dessa forma, o projeto consolida-se não apenas como uma prova de conceito funcional, mas como um modelo didático e replicável, capaz de inspirar soluções futuras na área de integração entre aplicações e bancos de dados.

6 REFERÊNCIAS

CASTELANO, C. R. **História dos Bancos de Dados**. [S. l.: s. n.], [s. d.] Disponível em: https://castelano.com.br/site/aulas/bd/Aula%2001%20-%20Introdu%C3%A7%C3%A3o.pdf. Acesso em: 11 maio 2025.

CHEN, P. P. S. **The Entity-Relationship Model:** Toward a Unified View of Data. ACM Transactions on Database Systems, v. 1, n. 1, p. 9–36, 1976. DOI: 10.1145/320434.320440

CODD, E. F. **A Relational Model of Data for Large Shared Data Banks.** Communications of the ACM, v. 13, n. 66, p. 377-387, 1970. DOI: 10.1145/362384.362685.

DATE, C. J. **An Introduction to Database Systems**. 8. ed. Boston: Addison-Wesley, 2003. Disponível em: https://lc.fie.umich.mx/~rodrigo/BD/An%20Introduction%20to%20Database%20Systems%208e%20By%20C%20J%20Date.pdf. Acesso em: 10 maio 2025.

DIAS, G.; OLIVEIRA, D. **O que é SQL?.** Alura**,** 2023. Disponível em: https://www.alura.com.br/artigos/o-que-e-sql. Acesso em: 11 set. 2024.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems.** 7. ed. [S. l.]: Pearson, 2016. Disponível em: https://auhd.edu.ye/upfiles/elibrary/Azal2020-01-22-12-28-11-76901.pdf. Acesso em: 10 maio 2025.

ESPOSITO, D. **Microsoft .NET:** Architecting Applications for the Enterprise. 2. ed. Redmond: Microsoft Press, 2018.

FOWLER, M. **Inversion of Control Containers and the Dependency Injection pattern**. 2004. Disponível em: https://martinfowler.com/articles/injection.html. Acesso em: 11 maio 2025.

GONÇALVES JUNIOR, A. C. **Um estudo de segurança da informação:** injeção de SQL. 2013. Instituto Municipal de Ensino Superior de Assis, Fundação Educacional do Município de Assis, Assis, 2013. Disponível em: https://cepein.femanet.com.br/BDigital/arqTccs/1111321076.pdf. Acesso em: 10 maio 2025.

MELTON, J.; SIMON, A. R. **SQL:** 1999 — Understanding Relational Language Components**.** San Francisco: Morgan Kaufmann, 2002.

MICROSOFT, 2024a. **Windows Presentation Foundation (WPF)**. Microsoft Learn, 2024. Disponível em: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/. Acesso em: 10 maio 2025.

MICROSOFT, 2024b. **Dependency Properties Overview**. Microsoft Learn, 2024. Disponível em: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/properties/dependency-properties-overview?view=netdesktop-9.0. Acesso em: 10 maio 2025.

MICROSOFT, 2024c. **.NET Architecture Guides**. Microsoft, 2024. Disponível em: https://dotnet.microsoft.com/en-us/learn/dotnet/architecture-guides. Acesso em: 10 maio 2025.

MICROSOFT, 2024d. **Dependency Injection in .NET**. Microsoft Learn, 2024. Disponível em: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. Acesso em: 10 maio 2025.

MOMJIAN, B. **PostgreSQL:** Introduction and Concepts. Boston: Addison-Wesley, 2001. Disponível em: https://lab.demog.berkeley.edu/Docs/Refs/aw_pgsql_book.pdf. Acesso em: 11 abril 2025.

POSTGRESQL GLOBAL DEVELOPMENT GROUP**. PostgreSQL Documentation – Version 16**. [S. l.]: PostgreSQL.org, 2024. Disponível em: https://www.postgresql.org/docs/current/index.html. Acesso em: 10 maio 2025.

PRICE, J. **Oracle Database 11g SQL.** Porto Alegre: Bookman, 2009. 684 p.

RICHTER, J. **CLR via C#**. 4. ed. Redmond: Microsoft Press, 2022.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts.** 7. ed. Nova York: McGraw-Hill, 2020.

STONEBRAKER, M. **SQL Databases v. NoSQL Databases.** Communications of the ACM, v. 53, n. 4, p. 10-11, 2010.

VISSOTTO JUNIOR, A. C.; BOSCO, E. B.; BRUSCHI, G. C.; SILVA, L. A. da. **Prevenção de Ataques:** XSS Residente e SQL Injection em Banco de Dados PostgreSQL em Ambiente WEB. Faculdade de Tecnologia de Bauru (FATEC), 2024. Disponível em: https://bkpsitecpsnew.blob.core.windows.net/uploadsitecps/sites/51/2024/09/1-66e2f70761a00.pdf. Acesso em: 10 maio 2025.

WIENER, R. Delegates and Events in C#. **Journal of Object Technology**, v. 3, n. 5, p. 78–85, maio-jun. 2004. Disponível em: https://www.jot.fm/issues/issue_2004_05/column8.pdf. Acesso em: 11 maio 2025.