Introdução

Dando continuidade à nossa série, nesta quarta parte vamos colocar em prática o que aprendemos até aqui, utilizando o pgAdmin 4 para executar scripts SQL e criar as primeiras tabelas no banco `dbdemo`. Além de entendermos como criar as tabelas pelo pgAdmin 4, vamos também explorar o que é um script **DDL** e qual o seu papel no contexto de bancos de dados relacionais.

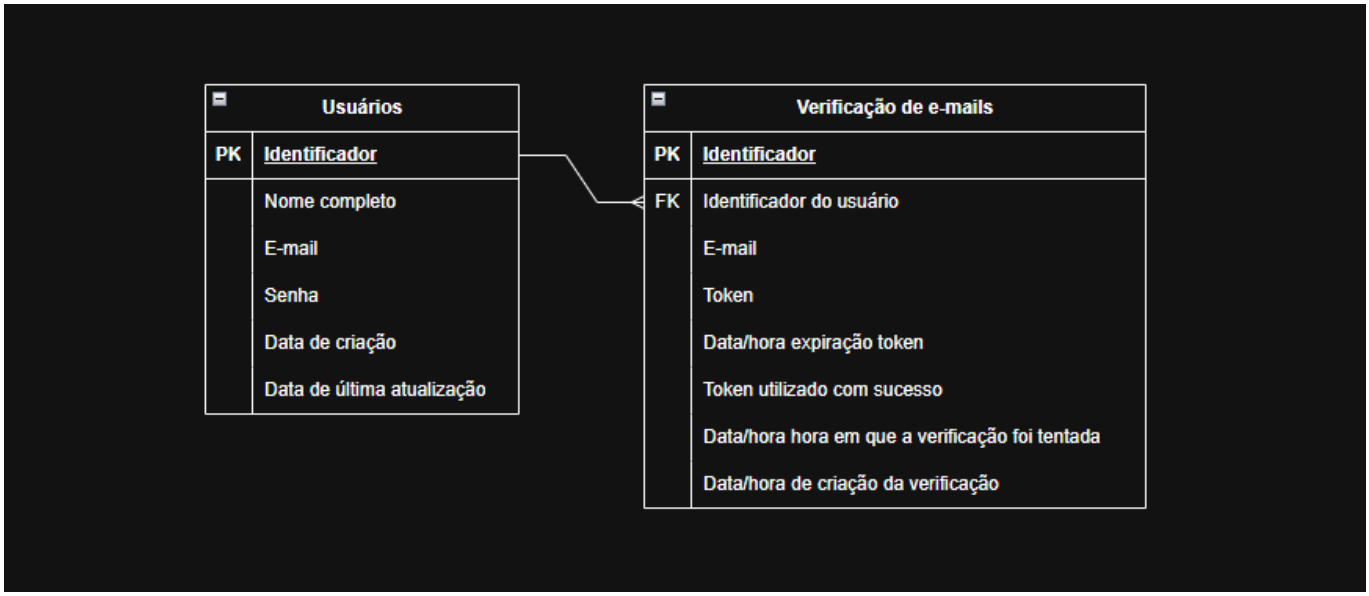
O que é um script DDL?

Em bancos de dados relacionais, como o PostgreSQL, os comandos SQL são divididos em categorias. Uma delas é chamada de DDL (Data Definition Language), ou *Linguagem de Definição de Dados*.

> ⚠**Nota:** DDL é a parte da linguagem SQL usada para definir a estrutura do banco de dados.

Modelagem Lógica

Antes de criar as tabelas no banco de dados é essencial planejar sua estrutura de forma organizada. Esse planejamento faz parte de um processo chamado modelagem de dados, e uma das suas etapas mais importantes é a modelagem lógica.



> **Atenção:** A modelagem lógica é a representação gráfica da estrutura de um banco de dados relacional, como o PostgreSQL.

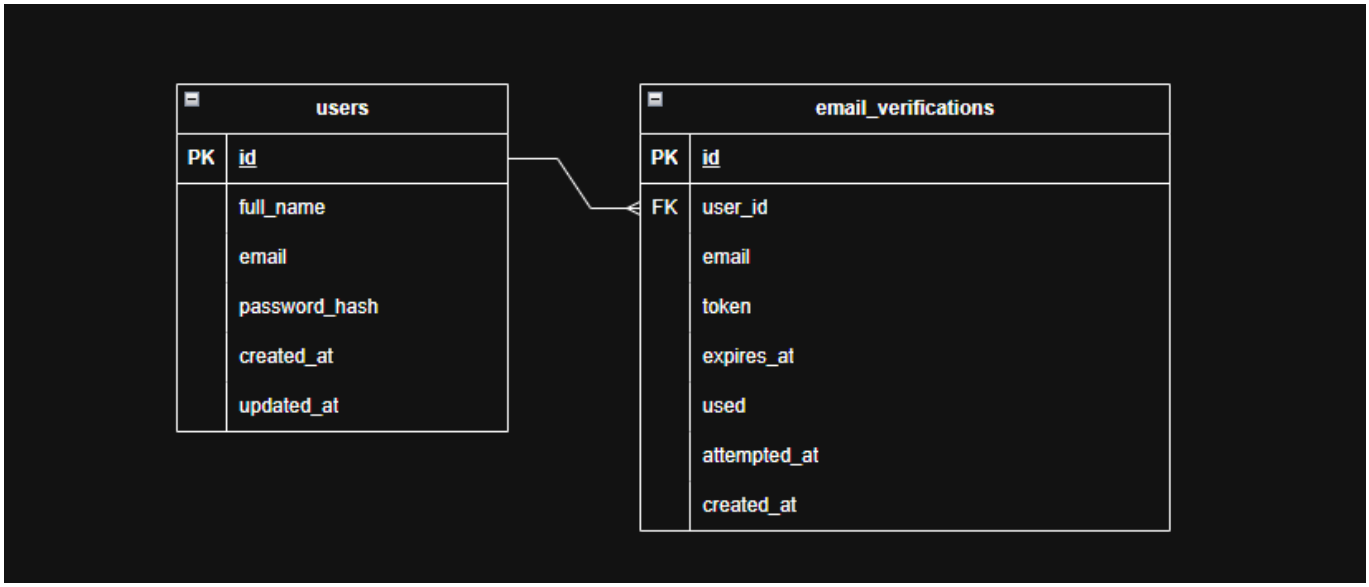
Ela descreve as tabelas , os campos que cada uma contém e os relacionamentos entre elas, sem ainda entrar nos detalhes técnicos da implementação. Essa visualização ajuda no entendimento, validação e comunicação da estrutura de dados com todos os envolvidos do projeto.

Além de ajudar a entender como os dados se organizam, a modelagem lógica também facilita a transição para a etapa seguinte: a criação física das tabelas no banco de dados. Para construir o diagrama apresentado, utilizamos a ferramenta gratuita draw.io, que permite criar diagramas ER (Entidade relacionamento) de forma simples e visual. Apesar disso, o objetivo aqui não é ensinar o uso da ferramenta, mas sim apresentar a estrutura planejada para o nosso sistema, de forma que qualquer pessoa consiga acompanhar o raciocínio mesmo sem conhecimentos técnicos avançados.

Modelagem Física

Depois de planejar a estrutura do banco de dados por meio da modelagem lógica, avançamos agora para a etapa de modelagem física, que é quando transformamos os conceitos e relações de modelo em estrutura real e executável dentro do banco de dados.

Enquanto a modelagem lógica foca em representar tabelas, campos e relacionamentos de forma visual e conceitual, a modelagem física trata da implementação prática no sistema de banco de dados escolhido, neste caso o **PostgreSQL**.



Atenção: Em outras palavras, a modelagem física é o que efetivamente cria as tabelas, define os tipos de dados, restrições, chaves primárias e estrangeiras, tudo com base no

modelo lógico definido anteriormente.

Para quem busca uma ferramenta que além de modelar também gere automaticamente os scripts DDL prontos para uso no PostgreSQL, uma boa alternativa gratuita é o DBeaver. Essa ferramenta trabalha exclusivamente com a modelagem física do banco de dados, ou seja, permite criar a estrutura real das tabelas, definir colunas, tipos de dados, chaves primárias e estrangeiras, e gerar o script SQL correspondente. Diferente de soluções voltadas à modelagem lógica, como o draw.io, o DBeaver não tem foco em abstrações conceituais, mas sim na implementação prática da estrutura que será executada no banco.

Scripts DDL

Agora que já passamos pelas etapas de modelagem lógica e modelagem física, chegou o momento de traduzir essa estrutura planejada em comandos SQL. O script abaixo representa a criação das tabelas `users` e `email_verifications` no PostgreSQL, utilizando instruções DDL (*Data Definition Language*), que são responsáveis por definir a estrutura real do banco de dados.

É importante destacar que, neste exemplo, os scripts foram escritos manualmente, pois utilizamos o draw.io apenas para a representação visual do modelo. Embora existam ferramentas que geram o DDL automaticamente a partir da modelagem, nossa proposta aqui é justamente reforçar o entendimento da estrutura por meio da prática com código real.

Tabela de Usuários

A tabela de `users` tem como objetivo armazenar os dados essenciais dos usuários autenticáveis da plataforma. Ela representa a estrutura base para identificação e login, contendo campos como nome completo, e-mail, senha criptografada e registros de data de criação e atualização. Essa tabela serve como ponto central de referência para outras entidades relacionadas, como o controle de verificação de e-mails, permissões e histórico de atividades.

Antes de analisarmos o script dessa tabela, é importante compreender alguns elementos técnicos usados na usa definição. O tipo UUID é utilizado para gerar identificadores únicos, ideal para uso como chave primária em sistemas distribuídos. O tipo VARCHAR é aplicado para campos de texto, como nome e e-mail, onde é possível definir um limite de caracteres. Já o TIMESTAMP é usado para registrar datas e horários, como os momentos de criação ou atualização de um registro.

```
CREATE TABLE users (
    id UUID PRIMARY KEY,
    full_name VARCHAR(150) NOT NULL,
    email VARCHAR(150) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP NOT NULL
);
```

Além disso, o campo `id` é marcado como PRIMARY KEY, garantindo que cada usuário tenha um identificador exclusivo. O campo `email`, por sua vez, está marcado como UNIQUE, o que impede que dois usuários diferentes utilizem o mesmo e-mail para cadastro no sistema.

Por fim, a maioria dos campos da tabela está marcada com a restrição NOT NULL, o que significa que esses valores são obrigatórios no momento do cadastro. Essa regra garante que os registros seja criados sempre com as informações essenciais preenchidas, evitando inconsistências e prevenindo erros durante o uso da aplicação.

Tabela de Verificação de E-mails

A tabela `email_verifications` tem como finalidade registrar os tokens de verificação enviados aos usuários durante o processo de cadastro. Cada registro representa uma tentativa de verificação de posse de um endereço de e-mail informado, com controle de validade, status de uso e vínculo com o usuário correspondente. Essa tabela permite que o sistema controle quem de fato confirmou o e-mail, além de possibilitar o reenvio ou invalidação de tentativas anteriores.

Antes de analisarmos o script dessa tabela, é importante compreender os elementos utilizados na sua definição. o tipo UUID é novamente utilizado como identificador único do registro, bem como para o campo `user_id`, que referencia o usuário responsável pela solicitação da verificação. O campo email utiliza VARCHAR com restrição UNIQUE para impedir duplicações de verificação para o mesmo endereço. O campo `token` também é único e tem por objetivo validar cada tentativa de forma segura e rastreável. Já o tipo TIMESTAMP é aplicado nos campos de data de expiração (`expires_at`), tentativa (`attempted_at`) e criação (`created_at`).

```
CREATE TABLE email_verifications (
```

```
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL,
    email VARCHAR(150) NOT NULL UNIQUE,
    token VARCHAR(255) NOT NULL UNIQUE,
    expires_at TIMESTAMP NOT NULL,
    used BOOLEAN DEFAULT FALSE,
    attempted_at TIMESTAMP,
    created_at TIMESTAMP NOT NULL,

    -- Chave estrangeira apontando para a tabela de usuários
    CONSTRAINT fk_user_invites_users
        FOREIGN KEY (user_id)
        REFERENCES users(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

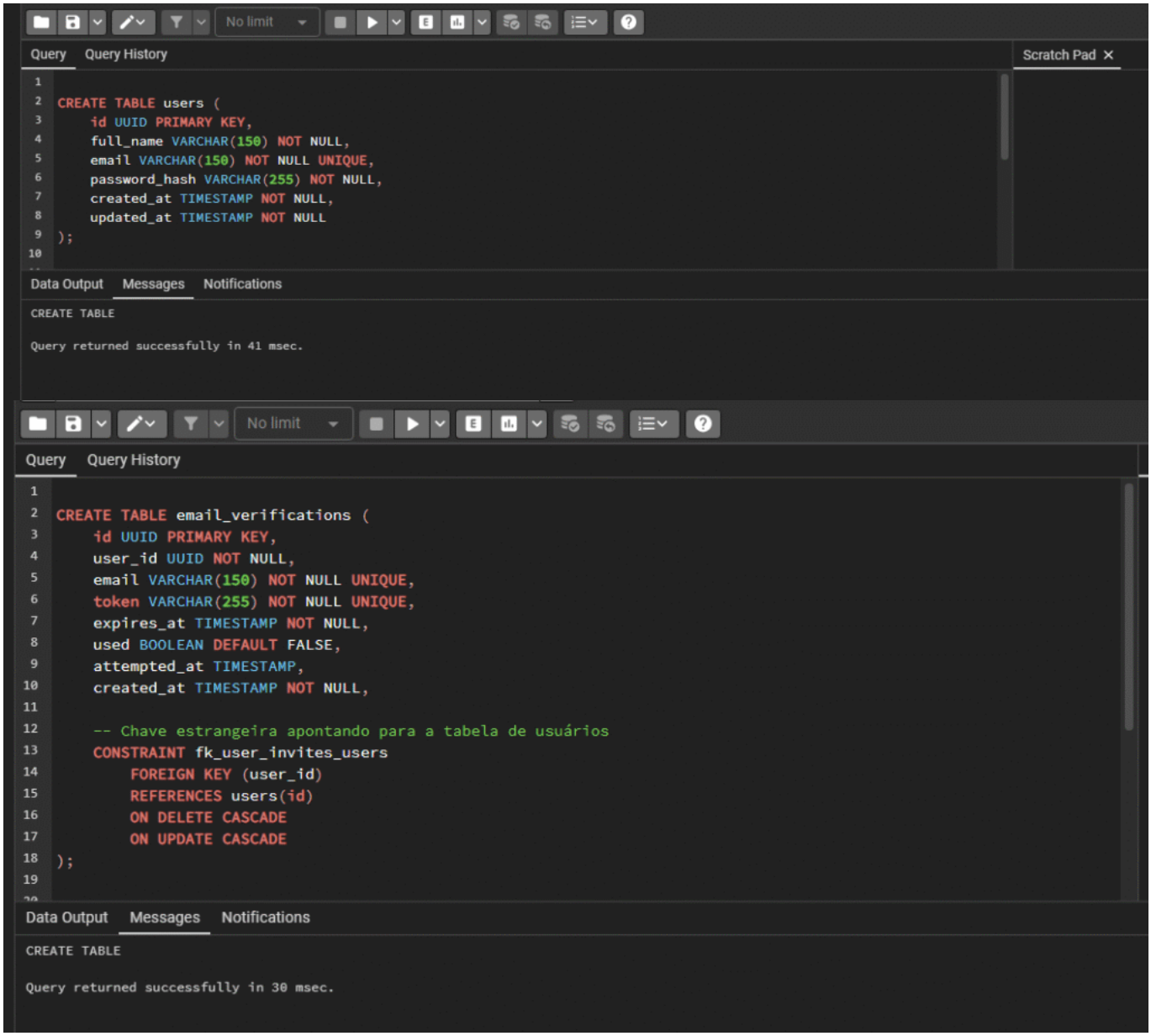
O campo `used` é um campo booleano com valor padrão `FALSE`, e serve para indicar se o token foi utilizado com sucesso. O campo `attempted_at`, por sua vez, é opcional e serve para registrar quando o usuário tentou utilizar o token, com ou sem êxito.

A tabela conta ainda com a restrição PRIMARY KEY no campo `id`, e com uma chave estrangeira no campo `user_id`, que faz referência à tabela `users`. Essa relação garante a integridade entre os dados do usuário e seus respectivos tokens de verificação. As cláusulas `ON DELETE CASCADE` e `ON UPDATE CASCADE` asseguram que, ao excluir ou atualizar um usuário, as verificações sejam automaticamente afetadas, evitando inconsistências.

Por fim, os campos com NOT NULL garantem que os registros sejam sempre criados com os dados mínimos necessários, reforçando a confiabilidade das validações e do processo de autenticação por e-mail.

Executando os scripts no pgAdmin 4

Agora que temos os scripts DDL completos e compreendemos a estrutura por trás de cada tabela, podemos seguir para a execução prática no banco de dados. Se você já estiver com o pgAdmin 4 instalado e conectado ao servidor PostgreSQL, basta abrir a ferramenta de consulta (Query Tool), colar os comandos apresentados neste e executá-los.



O pgAdmin irá processar os scripts e criar as tabelas `users` e `email_verifications` conforme modelado anteriormente. Em poucos segundos, sua estrutura estará pronta para ser visualizada e utilizada, respeitando exatamente o que foi definido na modelagem lógica, implementado na modelagem física e traduzido nos comandos SQL.

Visualizado as Tabelas Criadas no pgAdmin

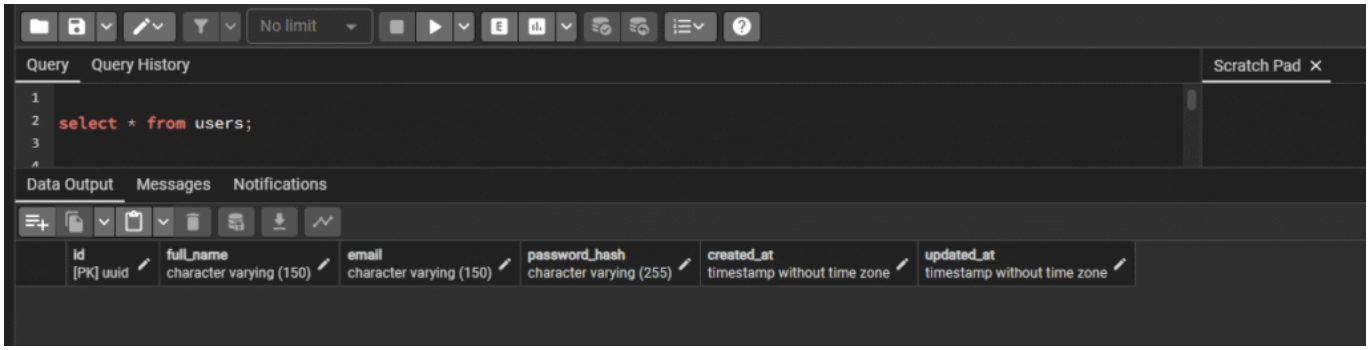
Após executar os scripts SQL no pgAdmin, conforme modelado anteriormente, as tabelas `users` e `email_verifications` são criadas com sucesso no banco de dados. Para validar essa criação, podemos

realizar consultas simples usando o comando SELECT e visualizar suas estruturas diretamente no painel de resultados do pgAdmin 4.

A primeira imagem mostra o resultado do comando abaixo, que consulta todos os registros da tabela `users`:

```
SELECT * FROM users;
```

Podemos observar que a estrutura da tabela `users` foi criada corretamente com as seguintes colunas:



- `id`: identificador único (UUID), chave primária;
- `full_name`: nome completo do usuário;
- `email`: endereço de e-mail;
- `password_hash`: senha criptografada;
- `created_at`: data de criação do registro;
- `updated_at`: data da última atualização.

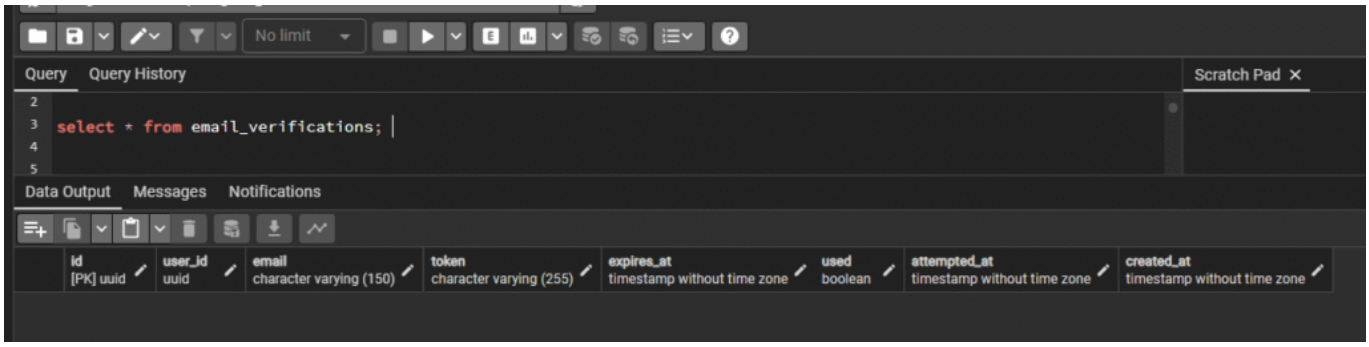
Essa estrutura garante que cada usuário seja identificado de forma única e que seus dados estejam organizados com segurança e rastreabilidade.

A segunda imagem apresenta o resultado da execução do comando abaixo, que retorna os dados da tabela email_verifications:

```
SELECT * FROM email_verifications;
```

Essa tabela foi criada com o objetivo de garantir a validade dos endereços de e-mail informados pelos usuários durante o processo de cadastro. Na prática, ela funciona como uma camada de segurança, antes que o sistema aceite definitivamente um novo registro, é enviado um e-mail para o endereço informado. Somente após a configuração, o cadastro é considerado válido. Dessa forma, evita que usuários mal-intencionados cadastrem endereços falsos ou inexistentes, preservando a integridade da base de dados.

- `id`: identificador único da verificação (UUID), chave primária;
- `user_id`: referência ao usuário (UUID);
- `email`: endereço de e-mail a ser verificado;
- `token`: código enviado por e-mail para validação;
- `expires_at`: data de expiração do token;
- `used`: indica se o token já foi utilizado (`boolean`);
- `attempted_at`: data da tentativa de verificação;
- `created_at`: data de criação do registro.



Com essa estrutura, o sistema controla todo o ciclo de verificação de e-mail, garantindo que apenas endereços reais sejam aceitos e que os tokens de confirmação tenham validade limitada.

Conclusão

Chegando ao fim desta série de quatro partes, conseguimos construir uma base sólida para qualquer desenvolvedor que deseje trabalhar com PostgreSQL de forma profissional e moderna no Windows, utilizando Docker no WSL2. Começamos pela instalação e preparação do ambiente, passamos pela criação automatizada do banco, integração com o pgAdmin e, por fim, desenvolvemos duas tabelas reais (`users` e `email_verifications`), aplicando conceitos de modelagem lógica, física e segurança de dados.

De forma resumida, vimos:

- **Parte 1**: Como instalar e preparar o ambiente com Docker e WSL2 para rodar o PostgreSQL com eficiência;

- **Parte 2**: Como configurar o banco `dbdemo` automaticamente com Dockerfile e Docker Compose, respeitando locale pt-BR e estrutura de pastas organizada;
- **Parte 3**: Como instalar e conectar o pgAdmin ao PostgreSQL no WSL2, criando uma interface gráfica prática para gestão do banco;
- **Parte 4**: Como criar tabelas reais com scripts DDL no pgAdmin, aplicando boas práticas de estrutura, validação e integridade de dados.

Este cenário que montamos será a base de dados utilizada nos projetos futuros com backend em .NET Core, onde utilizaremos as tabelas `users` e `email_verifications` como ponto de partida para implementar funcionalidades como autenticação de usuários, verificação de e-mails e acesso direto a dados em um banco relacional PostgreSQL.

Nos próximos conteúdos, vamos demonstrar como conectar uma aplicação .NET Core ao banco `dbdemo`, realizar consultas e interações reais com as tabelas já criadas, trazendo à prática o que construímos até aqui.