Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

VÍDEO DESTA AULA

Databases

Abaixo veremos duas formas de acessar um banco de dados PostgreSQL em Rust: primeiro sem utilizar um ORM (mapeamento objeto-relacional) e depois utilizando um ORM (neste caso, Diesel). Ambos os exemplos são simples e servem para demonstrar a conexão, inserção e consulta de dados em uma tabela chamada **pessoas**, com colunas **id** e **nome**.

Acesso ao PostgreSQL sem ORM

A seguir, temos um **exemplo completo** de acesso **síncrono** ao banco de dados PostgreSQL em Rust, **sem** usar `Tokio` ou ORM. Usamos a biblioteca `postgres` (que é bloqueante/síncrona) e uma tabela chamada `pessoas`:

Arquivo `Cargo.toml`

Crie (ou edite) o arquivo `Cargo.toml` na raiz do projeto, definindo o nome do pacote e adicionando a dependência:

```
[package]
name = "exemplo_postgres_sincrono_pt"
version = "0.1.0"
edition = "2021"

[dependencies]
postgres = "0.21"
```

Arquivo `main.rs`

No diretório `src`, crie (ou edite) um arquivo chamado `main.rs`:

```
use postgres::{Client, NoTls, Error};

fn main() -> Result<(), Error> {
    // 1. Conectar ao banco de dados de forma síncrona
    //    Ajuste a string de conexão conforme seu ambiente (host, user,
password, dbname).
    let mut cliente = Client::connect(
        "host=localhost user=postgres password=postgres
dbname=exemplo_rust",
        NoTls,
    )?;

    // 2. Criar a tabela "pessoas" se ela não existir
    //    Aqui, criamos duas colunas: id (chave primária) e nome (VARCHAR).
    cliente.execute(
        "CREATE TABLE IF NOT EXISTS pessoas (
            id SERIAL PRIMARY KEY,
            nome VARCHAR NOT NULL
        )",
        &[],
    )?;

    // 3. Inserir dados na tabela
    //    Usamos placeholders ($1) para evitar SQL injection.
    cliente.execute(
        "INSERT INTO pessoas (nome) VALUES ($1)",
        &[&"João da Silva"],
    )?;

    // 4. Consultar dados da tabela
    //    A função query retorna um vetor de linhas (Row).
    let linhas_encontradas = cliente.query("SELECT id, nome FROM pessoas",
&[])?;

    // 5. Exibir os resultados obtidos
    for linha in linhas_encontradas {
        // "linha.get(0)" retorna o valor da primeira coluna (id), do tipo
i32
        let identificador: i32 = linha.get(0);
        // "linha.get(1)" retorna o valor da segunda coluna (nome), do tipo
String
        let nome_da_pessoa: String = linha.get(1);

        println!("ID: {}, Nome: {}", identificador, nome_da_pessoa);
    }

    // 6. Retornar Ok(()) indicando sucesso
```

```
    Ok(())
}
```

Explicando o código

- **Linha 1**: Importamos `Client`, `NoTls` e `Error` da crate `postgres`.
    - `Client` é a estrutura principal para gerenciar a conexão síncrona.
    - `NoTls` indica que não estamos usando nenhuma camada de criptografia (TLS) adicional.
    - `Error` é o tipo de erro que pode ser retornado pelas funções de banco.
- **`fn main() -> Result<(), Error>`**: Definimos a função principal para retornar um `Result`; caso algo dê errado, retornamos um `Error` da crate `postgres`.
- **`Client::connect(...)`**: Cria uma conexão bloqueante (síncrona) com o Postgres usando as credenciais passadas na string. Se não for possível conectar, a função retorna um erro (`?` propaga o erro para `main`).
- **Criação da tabela**: Executamos um comando SQL simples, `CREATE TABLE IF NOT EXISTS pessoas (...)`, usando `cliente.execute(...)`. O retorno (número de linhas afetadas) não é tão relevante, por isso descartamos.
- **Inserção de dados**: Fazemos `INSERT INTO pessoas (nome) VALUES ($1)`, passando `&"João da Silva"` como parâmetro. Usar placeholders (`$1`, `$2`, etc.) ajuda a evitar SQL injection e facilita o binding de parâmetros.
- **Consulta**: Chamamos `cliente.query(...)` para executar `SELECT id, nome FROM pessoas`. Recebemos um vetor de linhas (Row).
- **Leitura de colunas**: Para cada linha, fazemos `linha.get(0)` ou `linha.get("nome")` para obter os dados. Mapeamos para `i32` (no caso do `id`) e `String` (para `nome`).
- **Exibição**: Imprimimos `ID: ..., Nome: ...` no console.
- **Encerramento**: Quando `main` termina, o objeto `cliente` é descartado, e a conexão é finalizada.

Como executar

1. **Instale o Rust** (via rustup.rs se ainda não o tiver).
2. **Instale o PostgreSQL** e crie um banco chamado `exemplo_rust` (ou ajuste a string de conexão no código).
3. Dentro da pasta do projeto, rode:

```
cargo run
```

4. Verifique no terminal se apareceu a mensagem “ID: 1, Nome: João da Silva” (caso a tabela estivesse vazia antes).

Isso demonstra o acesso **síncrono** ao PostgreSQL, **sem** usar `Tokio` (ou `async`/`await`) e **sem** um ORM (só consultas SQL puras).

Acesso ao PostgreSQL com ORM (Diesel)

Agora vamos usar o Diesel, um ORM para Rust. Ele gera e gerencia consultas em Rust de forma mais expressiva e segura em tempo de compilação.

Pré-requisitos

- **Instalar o CLI do Diesel** (opcional, mas recomendado) para facilitar a criação de *migrations*:

```
sudo apt-get update
sudo apt-get install -y build-essential libpq-dev pkg-config libssl-dev
cargo install diesel_cli --no-default-features --features postgres
```

- **Configurar o arquivo** `.env` com a URL de conexão ao seu banco de dados. Exemplo:

```
DATABASE_URL=postgres://postgres:postgres@localhost/exemplo_rust
```

`Cargo.toml`

```toml
[package]
name = "exemplo_postgres_orm"
version = "0.1.0"
edition = "2021"

[dependencies]
diesel = { version = "2.2.6", features = ["postgres"] }
dotenvy = "0.15"  # Usado para ler variáveis do .env
```

Estrutura de pastas

A estrutura típica de um projeto com Diesel (utilizando `diesel_cli`) inclui:

```
exemplo_postgres_orm
├── Cargo.toml
├── .env
├── migrations
│   └── YYYY-MM-DD-nnnn_nome_migration
│       ├── up.sql
│       └── down.sql
└── src
```

```
    ├── main.rs
    └── schema.rs
```

Arquivo de migration

Usando o Diesel CLI, podemos gerar uma migration "em branco":

```
diesel migration generate criar_pessoas
```

Isso criará uma pasta em `migrations` com timestamp, por exemplo:

```
migrations/
└── 2023-01-01-000000_criar_pessoas
    ├── up.sql
    └── down.sql
```

Editar `up.sql` e `down.sql`

No `up.sql` (migração "pra cima"), definimos a criação da tabela:

```sql
DROP TABLE IF EXISTS pessoas;
CREATE TABLE pessoas (
    id SERIAL PRIMARY KEY,
    nome VARCHAR NOT NULL
);
```

No `down.sql` (migração "pra baixo"), definimos como desfazer essa criação:

```sql
DROP TABLE pessoas;
```

Rodar a migration

Agora executamos:

```
diesel migration run
```

O Diesel vai:

1. Ler a variável `DATABASE_URL` do `.env`.
2. Criar a tabela `pessoas` se não existir (em `exemplo_rust`, no caso).

3. Criar uma tabela interna de controle de versões de migrations (chamada `__diesel_schema_migrations`, por padrão).

Verifique se a tabela foi criada, por exemplo, usando o `psql` ou outro cliente:

```
psql -h localhost -U postgres -d postgres -c "\dt"
```

Deverá constar `pessoas` lá.

Gerando o arquivo `schema.rs`

O Diesel mapeia as tabelas para código Rust via o "schema". Podemos gerar automaticamente o conteúdo do `schema.rs` com:

```
diesel print-schema > src/schema.rs
```

> Evite criar o `schema.rs` manualmente, pois perderá muito tempo.

Isso analisará o banco (via `DATABASE_URL`) e gerará algo assim:

```
// src/schema.rs

diesel::table! {
    pessoas (id) {
        id -> Int4,
        nome -> Varchar,
    }
}
```

> Este arquivo não deve ser editado manualmente se você usa `print-schema` regularmente. Caso contrário, pode manter manualmente sincronizado.

Criar o model (struct) em Rust

> O arquivo `schema.rs` descreve o mapeamento das tabelas do banco (estruturas e tipos de cada coluna) e normalmente é gerado automaticamente pelo Diesel. O arquivo `models.rs` define as structs Rust que representam as linhas dessas tabelas, por exemplo `Pessoa` e `NovaPessoa`, com `#[derive(Queryable)]` ou `#[derive(Insertable)]`, vinculando essas structs ao que foi definido em `schema.rs`.

**Por que duas `structs` para a mesma tabela?

> Precisamos de duas structs porque, ao **inserir** dados, nem sempre queremos (ou podemos) incluir valores que são gerados automaticamente (como o `id`), enquanto, ao **carregar** dados (fazer SELECT),

> precisamos de todos os campos que a tabela retorna. Assim, `NovaPessoa` normalmente omite campos auto-gerados, enquanto `Pessoa` representa a linha completa no banco, incluindo o `id`.

Na pasta `src`, crie (ou edite) um arquivo `models.rs` para colocar as estruturas que representam a tabela. Exemplo:

```
use super::schema::pessoas; // "super" pois o schema.rs está em src, mesmo nível

// Representação de uma linha na tabela (para SELECT).
// A trait Queryable permite ao Diesel "popular" esse struct ao fazer a query.
#[derive(Queryable)]
pub struct Pessoa {
    pub id: i32,
    pub nome: String,
}

// Representação dos dados que inserimos (INSERT).
#[derive(Insertable)]
#[diesel(table_name = pessoas)]
pub struct NovaPessoa<'a> {
    pub nome: &'a str,
}
```

> Note que `NovaPessoa` não tem `id`, pois o `id` é gerado automaticamente (SERIAL).

Editar o `main.rs`

Por fim, criamos o `main.rs` para:

1. Conectar ao banco usando Diesel.
2. Criar (inserir) registros na tabela `pessoas`.
3. Ler e exibir.

```
mod schema;
mod models;

use diesel::prelude::*;
use dotenvy::dotenv;
use std::env;

// Importa as coisas que vamos usar
use crate::models::{NovaPessoa, Pessoa};
use crate::schema::pessoas::dsl::*;

fn estabelecer_conexao() -> PgConnection {
    dotenv().ok(); // Lê variáveis do arquivo .env
    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL não definida no .env");
```

```
    PgConnection::establish(&database_url)
        .unwrap_or_else(|_| panic!("Falha ao conectar em {}",
database_url))
}

fn main() {
    // 1. Cria a conexão mutável
    let mut conexao = estabelecer_conexao();

    // 2. Insere uma nova pessoa
    let maria = NovaPessoa { nome: "Maria das Couves" };

    diesel::insert_into(pessoas)
        .values(&maria)
        // repare que passamos &mut conexao
        .execute(&mut conexao)
        .expect("Erro ao inserir pessoa");

    // 3. Consulta a tabela
    let resultado = pessoas
        .load::<Pessoa>(&mut conexao) // &mut conexao
        .expect("Erro ao carregar pessoas");

    println!("Lista de pessoas:");
    for p in resultado {
        println!("ID: {}, Nome: {}", p.id, p.nome);
    }
}
```

Executando o projeto

No terminal, dentro da pasta do projeto:

```
cargo run
```

- O Diesel lerá seu `.env` para conectar ao PostgreSQL.
- Executará `main.rs`, que insere “Maria das Couves” na tabela `pessoas`.
- Depois consulta a tabela e imprime o `id` e `nome` de cada registro no terminal.

Fluxo de criação resumido

1. **`Cargo.toml`**: inclua as dependências `diesel` e `dotenvy`.
2. **Instale Diesel CLI** (opcional, mas recomendado).
3. **Crie o `.env`** com `DATABASE_URL`.
4. **Crie as migrations** (`diesel migration generate <nome>`), escreva o SQL em `up.sql`/`down.sql`, e rode `diesel migration run`.

5. **Gere o schema** (`diesel print-schema > src/schema.rs`).
6. **Crie o model** (arquivos `models.rs`, definindo structs com `#[derive(Queryable)]`, `#[derive(Insertable)]`, etc.).
7. **Implemente o código** no `main.rs` (ou em outro binário) para usar Diesel:
    - Conectar ao DB,
    - Inserir registros,
    - Consultar,
    - Exibir resultados.