1. Introduction

rust\_orm\_gen is a Rust library designed to reverse engineer PostgreSQL databases and

automatically generate Rust structs and CRUD operations. This tool simplifies the process of

interacting with a PostgreSQL database in Rust, ensuring that your code is clean, maintainable, and

efficient.

2. Installation

Add rust\_orm\_gen to your Cargo.toml:

[dependencies]

rust\_orm\_gen = { path = "../path\_to\_your\_local\_crate" }

tokio = { version = "1", features = ["full"] }

dotenv = "0.15.0"

3. Configuration

Ensure your .env file is correctly configured with the database URL:

DATABASE\_URL=postgres://user:password@localhost/mydb

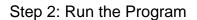
Replace user, password, and mydb with your actual PostgreSQL credentials and database name.

4. Usage

Step 1: Initialize the Database Context

Create a file named main.rs to run the reverse engineering tool:

```
mod context;
mod metadata;
mod generator;
mod crud;
mod db;
use crate::context::DbContext;
use dotenv::dotenv;
use std::env;
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
  dotenv().ok();
  let database_url = env::var("DATABASE_URL")?;
  let db_context = DbContext::new(&database_url).await?;
  let output_dir = "db";
  db_context.reverse_engineer(output_dir).await?;
  Ok(())
}
```



In your terminal, navigate to your project directory and run:

cargo run

# 5. Example Project

Here's a step-by-step example of how to use rust\_orm\_gen in your own project.

# 1. Create a New Project

cargo new my\_project

cd my\_project

# 2. Add Dependencies

Update the Cargo.toml file in your project:

# [dependencies]

```
rust_orm_gen = { path = "../path_to_your_local_crate" }
tokio = { version = "1", features = ["full"] }
dotenv = "0.15.0"
```

#### 3. Create .env File

Add your PostgreSQL connection string to a .env file: DATABASE\_URL=postgres://user:password@localhost/mydb 4. Set Up Main Function Create a main.rs file in the src directory: mod context; mod metadata; mod generator; mod crud; mod db; use crate::context::DbContext; use dotenv::dotenv; use std::env; #[tokio::main] async fn main() -> Result<(), Box<dyn std::error::Error>> { dotenv().ok(); let database\_url = env::var("DATABASE\_URL")?; let db\_context = DbContext::new(&database\_url).await?;

```
let output_dir = "db";

db_context.reverse_engineer(output_dir).await?;

Ok(())
}

5. Run the Program

cargo run
```

#### 6. Generated Code Structure

users\_crud.rs

After running the program, the generated ORM files will be saved in the db directory. For example, if you have a table named users, it will generate two files: users.rs and users\_crud.rs.

```
#[derive(Debug, Serialize, Deserialize)]
pub struct Users {
    #[serde(rename = "id")] pub id: i32,
    #[serde(rename = "first name")] pub first_name: String,
    #[serde(rename = "last name")] pub last_name: String,
}
```

```
use tokio_postgres::Client;
pub async fn create_users(client: &Client, entity: &Users) -> Result<Users, tokio_postgres::Error> {
  let row = client.query_one(
     "INSERT INTO users (id, "first name", "last name") VALUES ($1, $2, $3) RETURNING *",
     &[&entity.id, &entity.first_name, &entity.last_name]
  ).await?;
  Ok(Users {
     id: row.get("id"),
     first_name: row.get("first name"),
     last_name: row.get("last name"),
  })
}
pub async fn get_users(client: &Client, id: i32) -> Result<Users, tokio_postgres::Error> {
  let row = client.query_one(
     "SELECT * FROM users WHERE id = $1",
     &[&id]
  ).await?;
  Ok(Users {
     id: row.get("id"),
    first_name: row.get("first name"),
```

```
last_name: row.get("last name"),
  })
}
pub async fn update_users(client: &Client, entity: &Users) -> Result<Users, tokio_postgres::Error> {
  let row = client.query_one(
     "UPDATE users SET "first name" = $1, "last name" = $2 WHERE id = $3 RETURNING *",
     &[&entity.first_name, &entity.last_name, &entity.id]
  ).await?;
  Ok(Users {
     id: row.get("id"),
     first_name: row.get("first name"),
     last_name: row.get("last name"),
  })
}
pub async fn delete_users(client: &Client, id: i32) -> Result<u64, tokio_postgres::Error> {
  let result = client.execute(
     "DELETE FROM users WHERE id = $1",
     &[&id]
  ).await?;
  Ok(result)
}
```

#### 7. Integrating the Generated Code

To use the generated ORM code in your project: 1. Include the Generated Files In your main project file (e.g., main.rs): mod db { pub mod users; pub mod users\_crud; } 2. Use the Generated Code Use the generated code to interact with the database: use db::users::Users; use db::users\_crud::{create\_users, get\_users, update\_users, delete\_users}; use tokio\_postgres::Client; #[tokio::main] async fn main() -> Result<(), Box<dyn std::error::Error>> { dotenv().ok(); let database\_url = env::var("DATABASE\_URL")?; let (client, connection) = tokio\_postgres::connect(database\_url, tokio\_postgres::NoTls).await?;

```
tokio::spawn(async move {
  if let Err(e) = connection.await {
     eprintln!("connection error: {}", e);
  }
});
let new user = Users {
  id: 1,
  first_name: "John".to_string(),
  last_name: "Doe".to_string(),
};
let created_user = create_users(&client, &new_user).await?;
println!("Created user: {:?}", created_user);
let fetched_user = get_users(&client, 1).await?;
println!("Fetched user: {:?}", fetched_user);
let updated_user = Users {
  id: 1,
  first_name: "Jane".to_string(),
  last_name: "Doe".to_string(),
};
let updated_user = update_users(&client, &updated_user).await?;
```

```
println!("Updated user: {:?}", updated_user);
let rows_deleted = delete_users(&client, 1).await?;
println!("Deleted {} user(s)", rows_deleted);
Ok(())
```

}