**TTD Rust Sql Repository**

Introduction

[[Mufraggi Hugo](https://medium.com/@hmufraggi?source=post_page-----5d453c2d3e0e--------------------------------)](https://medium.com/@hmufraggi?source=post_page-----5d453c2d3e0e--------------------------------)

[Mufraggi Hugo](https://medium.com/@hmufraggi?source=post_page-----5d453c2d3e0e--------------------------------)

·

Follow

6 min read

·

4 days ago

63

1

We continue our exploration of Rust following the initial article on middleware management with Actix. I sought a way to simplify article writing and reuse code components. In my search, I stumbled upon hexagonal architecture.

Hexagonal architecture decomposes a system into multiple interchangeable and loosely coupled components, such as the application core, the database, the user interface, test scripts, or interfaces with other systems.

With hexagonal architecture, I can interchange components and transition from a REST API layer to a gRPC API layer seamlessly.

For this article, we will create and test the component related to the database. I have chosen an SQL database, PostgreSQL.

**Useful Link**

SQLX is the library that will help us connect and interact with the database.

All code produced will be available on this repository.

**The Contract**

The first step is to define the contract. This includes all the interfaces and types that will interact with our repository's external and internal aspects.

We need to define the following:

* Enum errors for each necessary action: Create, Get, Delete, etc.
* The DbUser structure, represents the user table in the database
* The Repository trait contains all the function definitions we want to use
* Our UserRepository structure implements the repository trait This gives us our Enums and Structs:

#[derive(Deserialize, Serialize, Debug, PartialEq)]  
pub enum InsertError {  
 Conflict,  
 Unknown,  
}  
pub enum FetchAllError {  
 Unknown,  
}  
#[derive(Deserialize, Serialize, Debug, PartialEq)]  
pub enum FetchOneError {  
 NotFound,  
 Unknown,  
}  
pub enum DeleteError {  
 NotFound,  
 Unknown,  
}  
pub struct PostgresRepository {  
 db\_pool: Option<Pool<Postgres>>,  
}  
#[derive(Deserialize, Serialize, Debug, PartialEq)]  
pub struct DbUser {  
 id: String,  
 first\_name: String,  
 last\_name: String,  
 birthday\_date: NaiveDate,  
 city: String,  
}

And for the different implementations:

impl PostgresRepository {  
 pub async fn new\_pool(url\_db: &str) -> Result<PostgresRepository, ()> {  
 let tmp = PgPool::connect(&url\_db).await;  
 match tmp {  
 Ok(value) => Ok(Self {  
 db\_pool: Some(value),  
 }),  
 Err(err) => Err(()),  
 }  
 }  
}  
#[async\_trait]  
pub trait Repository {  
 async fn insert(&self, user: DbUser) -> anyhow::Result<DbUser, InsertError>;  
 async fn fetch\_all(&self) -> anyhow::Result<Vec<DbUser>, FetchAllError>;  
 async fn get(&self, id: String) -> anyhow::Result<DbUser, FetchOneError>;  
 async fn update(  
 &self,  
 id: String,  
 new\_db\_user: DbUser,  
 ) -> anyhow::Result<DbUser, FetchAllError>;  
 async fn delete(&self, number: u32) -> anyhow::Result<(), DeleteError>;  
}  
impl Repository for PostgresRepository {  
 fn insert(self, user: DbUser) -> anyhow::Result<DbUser, InsertError> {  
 todo!()  
 }  
 fn fetch\_all(&self) -> anyhow::Result<Vec<DbUser>, FetchAllError> {  
 todo!()  
 }  
 fn get(&self, id: String) -> anyhow::Result<DbUser, FetchOneError> {  
 todo!()  
 }  
 fn update(&self, id: string, new\_db\_user: DbUser) -> anyhow::Result<DbUser, FetchAllError> {  
 todo!()  
 }  
 fn delete(&self, number: u32) -> anyhow::Result<(), DeleteError> {  
 todo!()  
 }  
}

**Test Drive Development 🚀**

Now that we have defined our contract, let’s implement our functions in TDD and fasten our seat belts. For article length reasons, I will only cover Create and Get, but don’t worry, everything will be available on GitHub.

**Preamble**

Sqlx needs you to create an environment variable DATABASE\_URL. For me, the URL of my database is as follows:

export DATABASE\_URL="postgres://postgres:somePassword@localhost:5432/postgres”

After that, you can execute this command:

cargo sqlx prepare

This command will create a file sqlx-data.json that will store queries, associated types, and enable type checking during compilation. While writing the code, I encountered an error related to the type of birthday\_date. This forced me to use the correct date type required by the database.

**Insert Test**

We start by coding our tests. Let’s define the behavior we want:

* Create works, and our function returns the DbUser.
* If there is an error, we want an InsertError. To represent these behaviors, we want to create a DbUser and simulate an error in user creation.

First Test ⚠️ The tests we will perform are asynchronous, so we need the [tokio::test] decorator.

We initialize the user at its creation and the expected user at the end of the test.

For borrowing reasons, I cloned the variables. I didn’t want to spend too much time fighting with the borrow checker.

#[tokio::test]  
async fn create\_works() {  
 let charset = "abcdefghijkl";  
 let user = DbUser {  
 id: Uuid::new\_v4().to\_string(),  
 last\_name: generate(6, charset),  
 first\_name: generate(6, charset),  
 city: generate(6, charset),  
 birthday\_date: NaiveDate::from\_ymd(2015, 3, 14),  
 };  
 let user\_res = DbUser {  
 id: user.id.clone(),  
 last\_name: user.last\_name.clone(),  
 first\_name: user.first\_name.clone(),  
 city: user.city.clone(),  
 birthday\_date: NaiveDate::from\_ymd(2015, 3, 14),  
 };

Now we want to initialize our repository and call insert. We retrieve and unwrap the result (equivalent to getting in Scala).

let url = "postgres://postgres:somePassword@localhost:5432/postgres";  
let repo = PostgresRepository::new\_pool(url).await.unwrap();  
let res = repo.insert(user).await;  
let user\_create = res.unwrap()

Then we run the test using the PartialEq trait of DbUser with:

assert\_eq!(user\_create.eq(&user\_res), true)

**Second test**

We want to create a first user and attempt to create a second user with the same id. Since the id is a primary key, it cannot be written to the database twice.

The only interesting part of the test is the check we will perform. We need to verify that we have an InsertError::Conflict.

let res = repo2.insert(user2).await;  
assert\_eq!(res.err().unwrap(), InsertError::Conflict)

**Insert**

The trait has defined the definition of the insert function.

The first thing we will do is to retrieve the db\_pool; we need to unwrap the db\_pool to interact with the database.

let db\_pool = self.db\_pool.as\_ref().unwrap();

For the next steps, we want to execute our query. For this, you can refer to the official repository and their example.

let rec = query!(  
 r#"  
 INSERT INTO users (id, first\_name, last\_name, birthday\_date, city)  
 VALUES ( $1, $2, $3, $4, $5) returning id  
 "#,  
 db\_user.id.to\_string(),  
 db\_user.first\_name,  
 db\_user.last\_name,  
 db\_user.birthday\_date,  
 db\_user.city  
)  
 .fetch\_one(db\_pool)  
 .await;

Sqlx is not an ORM, so we will write our queries in plain SQL. The first part is the SQL query, and the second is the arguments that will replace $1, $2, $3,$4, $5. We use fetch\_one by passing the db\_pool as an argument. Then, we “await” everything and retrieve the query's result.

Now that we have the result, we want to perform a match on rec. This will allow us to handle cases where rec is Ok or an Err easily. If rec is Ok, the value contains Id, and we will return the created DbUser; in case of an error, we will return our type error InsertError::Conflict.

match rec {  
 Ok(value) => {  
 Ok(db\_user)  
 }  
 Err(\_) => Err(InsertError::Conflict)  
}

**Get Test**

Having taken the time to detail everything for insert, I will be more expeditious for get.

**First test**

We want to create a user and then test get. For this, we will use the insert endpoint coded and tested previously.

let id = Uuid::new\_v4().to\_string();  
let user = DbUser {  
 id: id.clone(),  
 last\_name: generate(6, charset),  
 first\_name: generate(6, charset),  
 city: generate(6, charset),  
 birthday\_date: NaiveDate::from\_ymd(2015, 3, 14),  
};

We create the user and then insert it.

let mut repo = PostgresRepository::new\_pool(url).await.unwrap();  
repo.insert(user).await;

Then, we can test our function.

let repo2 = PostgresRepository::new\_pool(url).await.unwrap();  
let res1 = repo2.get(id).await.unwrap();  
assert\_eq!(user\_res.eq(&res1), true)

**Second test**

For the second test, we will test an arbitrary id and verify that we get the correct error message.

assert\_eq!(user\_response, FetchOneError::NotFound)

Get The definition of the GET function has been defined by the trait.

async fn get(&self, id: String) -> anyhow::Result<DbUser, FetchOneError>;

As with Insert, we want to retrieve the db\_pool.

let db\_pool = self.db\_pool.as\_ref().unwrap();

For GET, we will use another sqlx function, query\_as::<\_, DbUser>, with the expected type, DbUser. Fetch\_one allows returning a single value.

let rec = query\_as::<\_ DbUser>(  
 "SELET id, first\_name, last\_name, birthday\_date, city FROM users WHERE id = $1",  
)  
.bind(id)  
.fetch\_one(db\_pool)  
.await;

As with insert, we will use pattern matching to handle cases where res is Ok or Err easily. For the OK case, we will retrieve the query's result and reassign it to DbUser, returning the created DbUser.

match rec {  
 Ok(value) => {  
 Ok(DbUser {  
 id: value.id,  
 last\_name: value.last\_name,  
 first\_name: value.first\_name,  
 city: value.city,  
 birthday\_date: value.birthday\_date,  
 })  
 }  
 Err(\_) => Err(FetchOneError::NotFound),  
}

**Conclusion**

We have implemented Create and Get in TDD. The next steps — List, Update, and Delete — are on my repository.

In the next article, we will tackle the Domain part that links the API layer and the repository.

See you soon!

[Rust](https://medium.com/tag/rust?source=post_page-----5d453c2d3e0e---------------rust-----------------)

[Tdd](https://medium.com/tag/tdd?source=post_page-----5d453c2d3e0e---------------tdd-----------------)

[Sql](https://medium.com/tag/sql?source=post_page-----5d453c2d3e0e---------------sql-----------------)

[Software Development](https://medium.com/tag/software-development?source=post_page-----5d453c2d3e0e---------------software_development-----------------)

[Software Engineering](https://medium.com/tag/software-engineering?source=post_page-----5d453c2d3e0e---------------software_engineering-----------------)