**Rust multi module microservices Part 3 — Database**

[[](https://medium.com/@omprakashsridharan?source=post_page-----808bfddae8a8--------------------------------)](https://medium.com/@omprakashsridharan?source=post_page-----808bfddae8a8--------------------------------)

[Omprakash Sridharan](https://medium.com/@omprakashsridharan?source=post_page-----808bfddae8a8--------------------------------)

·

Follow

4 min read

·

Jun 27

1

Let’s start with our first common module which will be fairly basic and provide a good base to start dealing with our database. Open the Cargo.toml inside the database crate(folder) and add the dependencies like the one below.

[package]  
name = "database"  
version = "0.0.0"  
edition = "2021"  
  
# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html  
  
[dependencies]  
sea-orm = { workspace = true }  
tracing = { workspace = true }  
testcontainers = { workspace = true }

We are adding sea-orm and tracing which are needed for us to provide an implementation for creating a database connection and logging some information.

Now let’s start with the implementation of our library in src/ [lib.rs](http://lib.rs/). I have not followed all the best practices of proper file structuring for brevity but feel free to split it as you wish 😃

use sea\_orm::{ConnectOptions, Database, DatabaseConnection, DbErr};  
use std::time::Duration;  
use tracing::log;  
  
pub async fn get\_connection(database\_url: &str) -> Result<DatabaseConnection, DbErr> {  
 let mut opt = ConnectOptions::new(database\_url.to\_owned());  
 opt.max\_connections(100)  
 .min\_connections(5)  
 .connect\_timeout(Duration::from\_secs(10))  
 .acquire\_timeout(Duration::from\_secs(10))  
 .idle\_timeout(Duration::from\_secs(10))  
 .max\_lifetime(Duration::from\_secs(10))  
 .sqlx\_logging(true)  
 .sqlx\_logging\_level(log::LevelFilter::Info);  
 return Database::connect(opt).await;  
}

* The code imports necessary dependencies such as sea\_orm, tracing, and std::time::Duration.
* The function get\_connection is defined, which takes a database\_url parameter as input and returns a Result containing a DatabaseConnection or a DbErr (database error) if any occurs.
* Within the function, a ConnectOptions struct is created and initialized with the provided database\_url.
* Various connection options are set using methods such as max\_connections, min\_connections, connect\_timeout, acquire\_timeout, idle\_timeout, and max\_lifetime. These options define parameters like the maximum and minimum number of connections, connection timeouts, and the lifespan of connections.
* The sqlx\_logging method is used to enable SQL query logging, and sqlx\_logging\_level specifies the log level for the logging.
* Finally, the Database::connect method is invoked with the configured ConnectOptions to establish a database connection asynchronously using await. The resulting connection is returned as a Result.

In summary, this code sets up a database connection using SeaORM with customizable connection options, including connection pooling, timeouts, and logging.

Let’s write a test to validate our implementation. I will not write unit tests with mocks again for brevity but rather try to keep the test environment as real as possible. That is where [testcontainers](https://github.com/testcontainers/testcontainers-rs) come to our help 😎 Add the below test code to the same file [lib.rs](http://lib.rs/)

#[cfg(test)]  
mod tests {  
 use crate::get\_connection;  
 use sea\_orm::{ConnectionTrait, DatabaseBackend, QueryResult, Statement};  
 use testcontainers::{clients, images};  
  
 #[tokio::test]  
 async fn test\_database\_connection() {  
 let docker = clients::Cli::default();  
 let database = images::postgres::Postgres::default();  
 let node = docker.run(database);  
 let connection\_string = &format!(  
 "postgres://postgres:postgres@127.0.0.1:{}/postgres",  
 node.get\_host\_port\_ipv4(5432)  
 );  
 let database\_connection = get\_connection(connection\_string).await.unwrap();  
 let query\_res: Option<QueryResult> = database\_connection  
 .query\_one(Statement::from\_string(  
 DatabaseBackend::Postgres,  
 "SELECT 1;".to\_owned(),  
 ))  
 .await  
 .unwrap();  
 let query\_res = query\_res.unwrap();  
 let value: i32 = query\_res.try\_get\_by\_index(0).unwrap();  
 assert\_eq!(1, value);  
 }  
}

* The code is wrapped in a #[cfg(test)] attribute, indicating that it is specific to the test environment and should only be compiled and executed during testing.
* Various imports are made to include the necessary dependencies for testing: get\_connection function from the current crate, sea\_orm for database-related functionality, and testcontainers for managing test containers.
* The actual test function is defined using the #[tokio::test] attribute. This attribute indicates that the test function will run asynchronously using the Tokio runtime, which is commonly used for asynchronous testing in Rust.
* Inside the test function, a Docker client is created using testcontainers::clients::Cli::default().
* A PostgreSQL image is specified as the test container using images::postgres::Postgres::default(). We are using PostgreSQL because we configured SeaORM with that in the root Cargo.toml.
* The PostgreSQL container is started using (database), which returns a handle to the running container.
* The connection string for the PostgreSQL database is constructed by retrieving the host port of the running container and formatting it appropriately.
* The get\_connection function is called to establish a connection to the PostgreSQL database using the constructed connection string. The connection is awaited using await and unwrapped using unwrap().
* A query is executed on the database connection to retrieve the value 1 from the database. The query is constructed using Statement::from\_string and executed using query\_one. The result is awaited using await and unwrapped using unwrap().
* The retrieved query result is assigned to query\_res as an Option<QueryResult>.
* The value 1 is extracted from the query result using try\_get\_by\_index and assigned to the variable value.
* An assertion is made using assert\_eq! to compare the retrieved value with the expected value of 1. If the assertion fails, the test will fail.

In summary, this test code sets up a PostgreSQL test container, establishes a connection to it, executes a query, and verifies the result. It demonstrates a basic test case for checking the functionality of the get\_connection function and querying the database. We have completed our database module which abstracts the connection and can be reused in other crates. In the next article, let's add implementation to another crate.