Rust SQLx basics with SQLite: super easy how to

database

November 17, 2022

In this tutorial, we will learn the basics of using the SQLx crate with Rust and SQLite. The SQLx crate is an async pure Rust SQL crate featuring compiling time-checked queries. However, it is not an ORM. We will look at how to create an SQLite database and do SQL operations on it with SQLx.

After we complete this tutorial we will have a good understanding of the basics for using SQLite databases in our future Rust projects using SQLx.

The repository with the finished project can be found here: https://github.com/tmsdev82/sqlx-sqlite-basics-tutorial.

Another rust and database-related article can be found here: PostgreSQL database with Rust: basic how to.

Contents [hide]

What is SQLx

SQLx is an easy-to-use async SQL crate for Rust. It supports PostgreSQL, MySQL, SQLite, a checked queries, and is runtime agnostic (it works on ckends (native-tls, rustls).

Furthermore, SQLx is cross-platform, so it can compile anywhere Rust is supported, has built-in connection pooling, and a number of other features: SQLx.

What is SQLite

SQLite is an embedded SQL database engine. It runs serverless, meaning it reads and writes directly to ordinary disk files. The code for SQLite is in the public domain, so it is free for any use. SQLite is very compact, even with all features enabled the library size can be less than 750KiB. Also, the database file format is cross-platform. Therefore, it can be copied between different kinds of systems (ex. 32-bit and 64-bit systems).

For these reasons, SQLite is very popular as an application file format. For example, used on edge devices like mobile phones, tablets, and game consoles.

Project setup

Let's create our project using cargo `cargo new sqlx-sqlite-basics-tutorial`.

Then let's add dependencies required to use SQLx in our `Cargo.toml` file:

```
1.  [package]
2.  name = "sqlx-sqlite-basics-tutorial"
3.  version = "0.1.0"
4.  edition = "2021"
5.
6.  # See more keys and their definitions at https://doc.rust-
    lang.org/cargo/reference/manifest.html
7.
8.  [dependencies]
9.  sqlx = { version = "0.6.2", features = ["runtime-tokio-native-
    tls", "sqlite"]}
10. tokio = { version = "1.20.0", features = ["macros"]}
```

As this will be a simple and basic project we don't need a lot of dependencies:

- sqlx: T Rust SQL crate featuring compile-time checke PostgreSQL, MySQL, and SQLite. Here we're choosing the tokio runtime and SQLite features.

- tokio: An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications. The async-runtime that we will use for async SQL actions.

SQLx queries and operations basics

In this section, we will look at the basic operations and queries.

Creating an SQLite database

Let's write some code to create our initial database file in `main.rs`:

```
1.  use sqlx::{migrate::MigrateDatabase, Sqlite};
2.
3.  const DB_URL: &str = "sqlite://sqlite.db";
4.
5.  #[tokio::main]
6.  async fn main() {
7.      if !Sqlite::database_exists(DB_URL).await.unwrap_or(false) {
8.          println!("Creating database {}", DB_URL);
9.          match Sqlite::create_database(DB_URL).await {
10.             Ok(_) => println!("Create db success"),
11.             Err(error) => panic!("error: {}", error),
12.         }
13.     } else {
14.         println!("Database already exists");
15.     }
16. }
```

First, we bring some items into scope: `MigrateDatabase` and `Sqlite`. Where the former (`MigrateDatabase`) is a trait that has functions: `create_database`, `database_exists`, and `drop_database`. We have to bring these into scope to be able to call them on `Sqlite`. `Sqlite` represents the database driver.

On line 3 we define the connection string for our SQLite database. This also determines the name of the database file that will be created.

Line 5: ar , so we have to declare the main function as `async`

Then on line 7, we check if the database exists, if an error occurs we return `false` with `unwrap_or`. If the database does not exist we create it on line 9 with `Sqlite::create_database(DB_URL)`. Next, we check for error or success with a `match` statement.

Running the program should look like this:

```
❯ cargo run --release
    Finished release [optimized] target(s) in 0.05s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Creating database sqlite://sqlite.db
Create db success
```

Create database success

After running the code, a new file should appear in our project's root directory: `sqlite.db`.

Creating a table with SQLx using Rust code

There are a number of ways to create a table with SQL. For example, using a raw SQL query in the Rust code or using a SQL migration script. First, we will use a query in Rust code. In a later section, we will look at using migration scripts.

To perform queries on the database we first have to connect to it, of course. So, let's use `SqlitePool` to create a pool object for connections. Then use that to execute a `CREATE TABLE` query:

```
1.  use sqlx::{migrate::MigrateDatabase, Sqlite, SqlitePool};
2.
3.  const DB_URL: &str = "sqlite://sqlite.db";
4.
5.  #[tokio::main]
6.  async fn main() {
7.      if !Sqlite::database_exists(DB_URL).await.unwrap_or(false) {
8.          println!("Creating database {}", DB_URL);
9.          match Sqlite::create_database(DB_URL).await {
10.             Ok(_) => println!("Create db success"),
11.             Err(error) => panic!("error: {}", error),
12.
13.
14.                                    eady exists");
15.
```

```
16.
17.     let db = SqlitePool::connect(DB_URL).await.unwrap();
18.
19.     let result = sqlx::query("CREATE TABLE IF NOT EXISTS users
    (id INTEGER PRIMARY KEY NOT NULL, name VARCHAR(250) NOT
    NULL);").execute(&db).await.unwrap();
20.     println!("Create user table result: {:?}", result);
21.  }
```

As mentioned, we create a connection pool using the `DB_URL` string on line 17. This `SqlitePool::connect` call returns `Pool<Sqlite>`.

We use a reference to this object when executing the `CREATE TABLE` query on line 19.

Running the program should result in output like this:

```
    Finished release [optimized] target(s) in 2.49s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Database already exists
Create user table result: SqliteQueryResult { changes: 0, last_insert_rowid: 0 }
```

The result doesn't really tell us much, other than that the query was successful. We can use a query on the table schema (`sqlite_schema`) to reveal all the tables in the database:

```
1.  use sqlx::{migrate::MigrateDatabase, Row, Sqlite, SqlitePool};
2.
3.  const DB_URL: &str = "sqlite://sqlite.db";
4.
5.  #[tokio::main]
6.  async fn main() {
7.      if !Sqlite::database_exists(DB_URL).await.unwrap_or(false) {
8.          println!("Creating database {}", DB_URL);
9.          match Sqlite::create_database(DB_URL).await {
10.             Ok(_) => println!("Create db success"),
11.             Err(error) => panic!("error: {}", error),
12.         }
13.     } else {
14.         println!("Database already exists");
15.     }
16.
17.     let db = SqlitePool::connect(DB_URL).await.unwrap();
18.
19.                                  'CREATE TABLE IF NOT EXISTS users
    (i                               LL, name VARCHAR(250) NOT
    NU                               wrap();
20.                                  e result: {:?}", result);
21.
```

```
22.         let result = sqlx::query(
23.             "SELECT name
24.              FROM sqlite_schema
25.              WHERE type ='table'
26.              AND name NOT LIKE 'sqlite_%';",
27.         )
28.         .fetch_all(&db)
29.         .await
30.         .unwrap();
31.
32.         for (idx, row) in result.iter().enumerate() {
33.             println!("[{}]: {:?}", idx, row.get::<String, &str>
    ("name"));
34.         }
35.     }
```

Because we want to get column values from the row on line 33 using `get`, we have to bring `Row` into scope on line 1.

So, by querying the table `sqlite_schema` for items with type `table` we can get the names of all the tables present in the database.

On lines 32-34 we display them by looping through the resulting `Vec<SqliteRow>`. Using `.iter().enumerate()` we can get the value and also the index number.

Finally, we get the value of the column on line 33 using `get`. We can use a string (`&str`) to index the columns and retrieve the value as a `String`. We have to specify the types here because `get` is a function that uses generics.

Now let's run the program again:

```
    Finished release [optimized] target(s) in 3.31s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Database already exists
Create user table result: SqliteQueryResult { changes: 0, last_insert_rowid: 0 }
[0]: "users"
```

Convert query result into data struct

In this se n. However, instead of using the generic `get()` fu the result into an object. We will write a

struct ourselves for this.

Defining the users data struct

Let's start by defining the data struct for the `users` table. This table only has two columns so it is very simple:

```
use sqlx::{migrate::MigrateDatabase, FromRow, Row, Sqlite, SqlitePool};

const DB_URL: &str = "sqlite://sqlite.db";

#[derive(Clone, FromRow, Debug)]
struct User {
    id: i64,
    name: String,
}
```

We use the `derive` macro here to implement the `FromRow` trait. This trait will allow us to use `query_as` to get results as the data struct we want. We also use `Clone` for making copies and `Debug` for easy display as debug information when needed.

Inserting user data, querying and mapping results to a type

Now let's insert some user data and then query it using `query_as` and our `User` struct. We'll add the following code below the `for` loop listing the tables:

```
    let result = sqlx::query("INSERT INTO users (name) VALUES (?)")
        .bind("bobby")
        .execute(&db)
        .await
        .unwrap();

    println!("Query result: {:?}", result);

    let user_results = sqlx::query_as::<_, User>("SELECT id, name FROM users")
        .fetch_all(&db)
        .await
        .unwrap();

    for user in user_results {
        println!("[{}, {}]", user.id, &user.name);
    }
```

First, we insert a new user record using a parameterized query on lines 42-46. We use a `?` symbol in the query here. However, it could also be `$` with a number, for example in this case it would be `$1`. If we use a parameter in the query we also have to bind a value to it with `bind()`, as seen on line 43.

Next, we print the result from the query execution on line 48.

Then, we use `query_as` on lines 50-53 to query the `users` table and map the results to a concrete type.

Finally, we print the results using the fields of the `User` struct which is much more convenient than using `get()` on the row object.

Running the program results in an output like this:

```
    Finished release [optimized] target(s) in 2.73s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Creating database sqlite://sqlite.db
Create db success
Create user table result: SqliteQueryResult { changes: 0, last_insert_rowid: 0 }
[0]: "users"
Query result: SqliteQueryResult { changes: 1, last_insert_rowid: 1 }
[1] name: bobby
```

Deleting records

While we're at it, let's also add a delete query after the insertion query:

```
59.         let delete_result = sqlx::query("DELETE FROM users WHERE
    name=$1")
60.             .bind("bobby")
61.             .execute(&db)
62.             .await
63.             .unwrap();
64.
65.         println!("Delete result: {:?}", delete_result);
```

Nothing new here, we're simply executing another query. Only this time we use `$1` to mark the parameter.

SQLx migrations with SQL and Rust code

So far we have done everything with just Rust code for our SQLx basics with SQLite project. When it comes to creating and updating tables it might be more convenient to make use of the migration mechanism.

Migrations or schema migrations are scripts for updating a database to the desired state. This could be by adding tables, or columns, even removing columns, or tables, changing column types, etc.

Installing the SQLx CLI

To use migrations we have to install the SQLx CLI tool: `cargo install sqlx-cli`. This will install the command line tool globally.

Adding a migration script

First, let's remove our current database file `sqlite.db` and also any other database files like `sqlite.db-shm` and `sqlite.db-wal`.

Then, let's add a migration using the following command from the command prompt in the root of our project directory: `sqlx migrate add users`. This will create a directory `migrations` and a file with a timestamp prefix and ending in `_users.sql`. The timestamp tells the migration code in what order to execute the scripts.

Currently, the file just has a comment line `-- Add migration script here`. So let's open it and add the following script:

```
1.  CREATE TABLE IF NOT EXISTS users
2.  (
3.      id          INTEGER PRIMARY KEY NOT NULL,
4.      name        VARCHAR(250)        NOT NULL,
5.                                      NOT NULL DEFAULT 0
6.  )
```

Applying migrations in Rust code

Now let's up data our Rust code to use the migration scripts instead of a `CREATE TABLE` query in code. We should also update our `User` struct to include the new `active` column:

```
use sqlx::{migrate::MigrateDatabase, FromRow, Row, Sqlite, SqlitePool};

const DB_URL: &str = "sqlite://sqlite.db";

#[derive(Clone, FromRow, Debug)]
struct User {
    id: i64,
    name: String,
    active: bool,
}

#[tokio::main]
async fn main() {
    if !Sqlite::database_exists(DB_URL).await.unwrap_or(false) {
        println!("Creating database {}", DB_URL);
        match Sqlite::create_database(DB_URL).await {
            Ok(_) => println!("Create db success"),
            Err(error) => panic!("error: {}", error),
        }
    } else {
        println!("Database already exists");
    }

    let db = SqlitePool::connect(DB_URL).await.unwrap();

    let crate_dir = std::env::var("CARGO_MANIFEST_DIR").unwrap();
    let migrations = std::path::Path::new(&crate_dir).join("./migrations");

    let migration_results = sqlx::migrate::Migrator::new(migrations)
        .await
        .unwrap()
        .run(&db)
        .await;

    match migration_results {
        Ok(_) => println!("Migration success"),
        Err(error) => panic!("error: {}", error),
    }
```

```
42.       println!("migration: {:?}", migration_results);
43.
44.       let result = sqlx::query(
45.           "SELECT name
46.            FROM sqlite_schema
47.            WHERE type ='table'
48.            AND name NOT LIKE 'sqlite_%';",
49.       )
50.       .fetch_all(&db)
51.       .await
52.       .unwrap();
53.
54.       for (idx, row) in result.iter().enumerate() {
55.           println!("[{}]: {:?}", idx, row.get::<String, &str>
    ("name"));
56.       }
57.
58.       let result = sqlx::query("INSERT INTO users (name) VALUES
    (?)")
59.           .bind("bobby")
60.           .execute(&db)
61.           .await
62.           .unwrap();
63.
64.       println!("Query result: {:?}", result);
65.
66.       let user_results = sqlx::query_as::<_, User>("SELECT id,
    name, active FROM users")
67.           .fetch_all(&db)
68.           .await
69.           .unwrap();
70.
71.       for user in user_results {
72.           println!(
73.               "[{}] name: {}, active: {}",
74.               user.id, &user.name, user.active
75.           );
76.       }
77.
78.       let delete_result = sqlx::query("DELETE FROM users WHERE
    name=$1")
79.           .bind("bobby")
80.           .execute(&db)
81.           .await
82.           .unwrap();
83.
84.       println!("Delete result: {:?}", delete_result);
85.   }
```

We have updated the `User` struct. But more importantly, on lines 26-42 we call code for performing database migrations:

```
26.       let crate_dir = std::env::var("CARGO_MANIFEST_DIR").unwrap();
27.       let migrations =
    std::path::Path::new(&crate_dir).join("./migrations");
28.
29.       let migration_results =
    sqlx::migrate::Migrator::new(migrations)
30.           .await
31.           .unwrap()
32.           .run(&db)
33.           .await;
34.
35.       match migration_results {
36.           Ok(_) => println!("Migration success"),
37.           Err(error) => {
38.               panic!("error: {}", error);
39.           }
40.       }
41.
42.       println!("migration: {:?}", migration_results);
```

In this code block we first get the path to the migrations directory by getting the root directory from the `CARGO_MANIFEST_DIR` environment variable. Then we join the `./migrations` directory path onto that.

Next on lines 29-33 we create a new instance of the `Migrator` and call `run(&db)` right away to start the migrations process.

Finally, we print the result of the migration on lines 35-42.

Updating users query

We should also update our `SELECT` query and printing of the results of the query:

```
66.       let user_results = sqlx::query_as::<_, User>("SELECT id,
    name, active FROM users")
67.           .fetch_all(&db)
68.           .await
69.
70.
71.
72.
```

```
            "[{}] name: {}, active: {}",
            user.id, &user.name, user.active
        );
    }
```

Here we added the `active` column in the query as well as the in the loop printing the results. We have to add the column in the query when we deserialize to the `User` object, because the field is not defined as optional using `Option<T>`.

Result of running the program

Running the program now results in the following output:

```
    Finished release [optimized] target(s) in 2.50s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Creating database sqlite://sqlite.db
Create db success
Migration success
migration: Ok(())
[0]: "_sqlx_migrations"
[1]: "users"
Query result: SqliteQueryResult { changes: 1, last_insert_rowid: 1 }
[1] name: bobby, active: false
Delete result: SqliteQueryResult { changes: 1, last_insert_rowid: 1 }
```

We can see that the migration is successful and that users table is again in the `sqlite_schema`. Furthermore, there is another table listed as well: the `_sqlx_migrations` table. This is where the system registers what migrations have been performed.

Adding another table via migration

Adding another table is of course simple in the same way we added the users table using the command: `sqlx migrate add items`.

This will add another file in the `migrations` directory with the suffix `_items.sql`. Let's add

```
C                                    ems
(
                                     RY KEY NOT NULL,
```

```
    name            VARCHAR(250)            NOT NULL,
    price           FLOAT                   NOT NULL DEFAULT 0
);
```

Updating a table

Of course, we can also update a table to add a new column. For example, to add a lastname to the users table. Let's add a migration script for that using the command `sqlx migrate add users_lastname`. And then add the following script to the file with the `_users_lastname.sql` suffix:

```
ALTER TABLE users  ADD lastname VARCHAR(250) NOT NULL DEFAULT 'unknown';
```

With this we need to update the code in the following way:

```
#[derive(Clone, FromRow, Debug)]
struct User {
    id: i64,
    name: String,
    lastname: String,
    active: bool,
}
```

Update the select query and printing:

```
    let result = sqlx::query("INSERT INTO users (name, lastname) VALUES (?,?)")
        .bind("bobby")
        .bind("fischer")
        .execute(&db)
        .await
        .unwrap();

    println!("Query result: {:?}", result);

    let user_results = sqlx::query_as::<_, User>("SELECT id, name, lastname, active FROM users")
        .fetch_all(&db)
        .await
        .unwrap();

    
    
    astname: {}, active: {}",
    e, &user.lastname, user.active
    ));
```

```
78.         }
```

And the result:

```
    Finished release [optimized] target(s) in 3.33s
     Running `target/release/sqlx-sqlite-basics-tutorial`
Database already exists
Migration success
migration: Ok(())
[0]: "_sqlx_migrations"
[1]: "users"
[2]: "items"
Query result: SqliteQueryResult { changes: 1, last_insert_rowid: 1 }
[1] name: bobby, lastname: fischer, active: false
Delete result: SqliteQueryResult { changes: 1, last_insert_rowid: 1 }
```

Conclusion

In this simple and quick tutorial, we learned the basics of using the SQLx crate and creating a SQLite database. We learned a little bit about migrations and parameterized queries as well. Now we have a foundation for writing simple applications that make use of a database for information storage.

The repository with the finished project can be found here: https://github.com/tmsdev82/sqlx-sqlite-basics-tutorial.

Please follow me on Twitter to get updates on more Rust programming-related tutorials: Follow @tmdev82

database rust sqlite sqlx

Comments (5)

Anonymous says:

April 22, 2023 at 9:11 am

> To use QLx CLI tool: cargo install sql-cli. This will install the

It should be `cargo install sqlx-cli`

Reply

Tim says:

April 28, 2023 at 3:45 am

Thanks for the correction.

Reply

Simon says:

August 1, 2023 at 2:39 pm

Hi Tim!

Great blog post, it helped me a lot.

Reply

Tim says:

August 24, 2023 at 2:47 am

I appreciate the message. I'm glad the blog post was helpful.

Reply

Michael Bushey says:

November 14, 2023 at 12:14 am

Great tutorial, it was up to date and nicely presented. Thank you. 🙂

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

Next Post

Solana wallet with Rust: get started now

Prev Post

Solana transactions per second: how to with Rust

Site Search

Follow @tmdev82

Recent Posts

Create your first React web app: easy basics

Python Virtual Environment (venv): easy guide for beginners

How to install node version manager (nvm)

Solana wa

Thank you for visiting. You can now buy me a coffee!

Rust SQLx basics with SQLite: super easy how to

Recent Comments

Tim on How to use JWT with Rust: learn the basics here
@nobody on How to use JWT with Rust: learn the basics here
eclipse on Basic how to log to a file in Rust with log4rs
Aleph on Plot Candles and SMA with Rust: learn how to
Michael Bushey on Rust SQLx basics with SQLite: super easy how to

Thank you for visiting. You can now buy me a coffee!

Thank you for visiting. You can now buy me a coffee!