



PV281: Programování v Rustu

Obsah

1. Docker pro setup prostředí
2. Postgres
3. SQLX
4. ORM & Diesel

Docker

Kontejnerizační technologie, kterou si můžeme představit jako lehkou virtualizaci.

Kontejner je standardizovaný balík softwaru, který poskytuje osekáný OS, knihovny potřebné pro běh aplikace a appku samotnou.

Instalace

stáhnout Docker for Desktop

<https://www.docker.com/get-started>

na Win10/11 je nutné nainstalovat nejdříve WSL2. Alternativa je mít nainstalované Hyper-V.

Práce s Dockerem

spustit CMD a vyzkoušet

```
docker run -dp 80:80 docker/getting-started
```

to se připojí na Docker Hub, stáhne image, spustí démona,
namapuje porty

Závislosti pro dev

```
# Use postgres/example user/password credentials
version: '3.1'

services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: example
    volumes:
      - db-data:/var/lib/postgresql/data

  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080

volumes:
  db-data:
```

PlantUML

```
@startuml
' hide the spot
hide circle

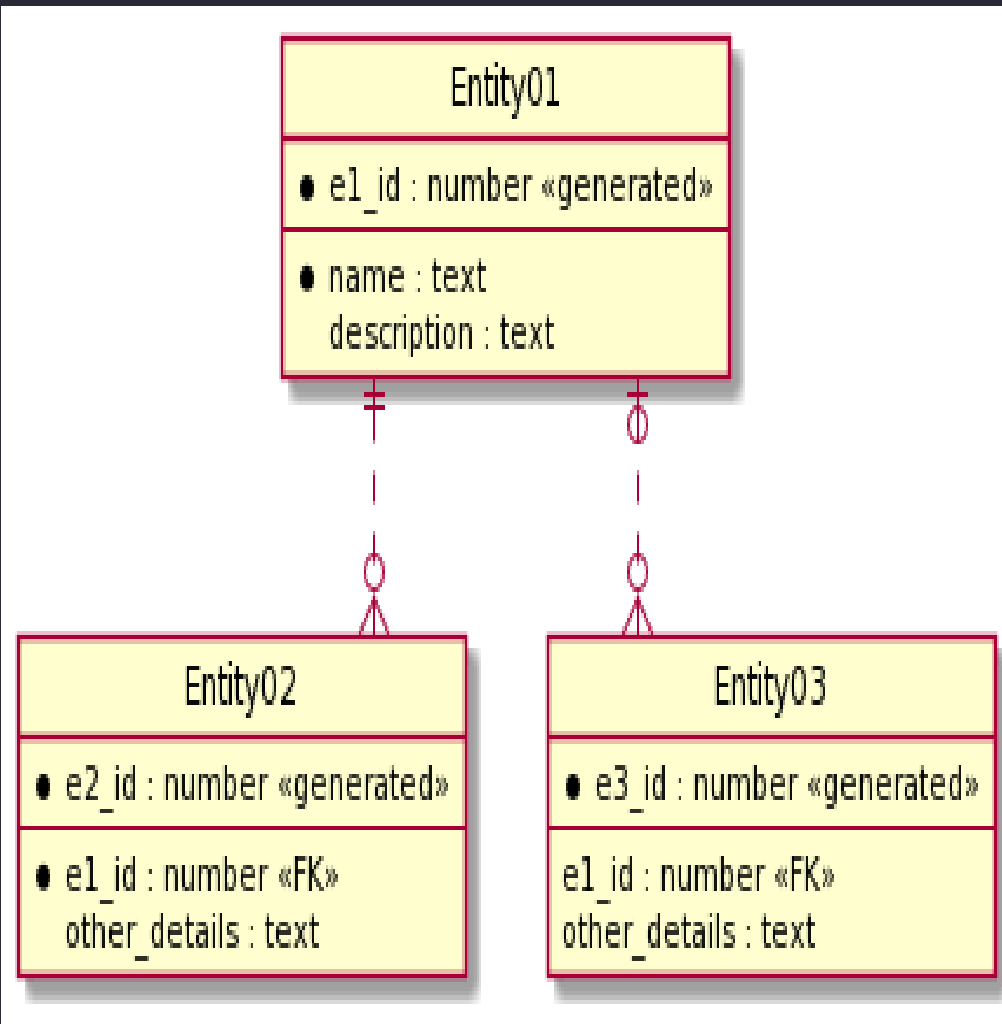
' avoid problems with angled crows feet
skinparam linetype ortho

entity "Entity01" as e01 {
  *e1_id : number <<generated>>
  --
  *name : text
  description : text
}

entity "Entity02" as e02 {
  *e2_id : number <<generated>>
  --
  *e1_id : number <<FK>>
  other_details : text
}

entity "Entity03" as e03 {
  *e3_id : number <<generated>>
  --
  e1_id : number <<FK>>
  other_details : text
}

e01 ||..o{ e02
e01 |o..o{ e03
@enduml
```



Postgres

klasická relační SQL databáze

open-source

s velkým množstvím funkcí

velmi dobrý výkon i pro velké systémy

ORM

= object relation mapping

V Rustu sice nemáme objekty, ale mapujeme na struktury.

Důležité detaily k ORM:

- omezuje množství možných útoků
- jednoduchá a na vývoj rychlá práce s databází
- vše typované, a tím pádem možné odhalit chyby
- ne vše podporované ORM knihovnamí
- vygenerované dotazy nemusí být ideální

Diesel

nejpoužívanější ORM v Rustu (ne že by bylo moc alternativ)
jeden z nejrychlejších ORM systémů
eliminuje runtime errorry při práci s DB (aspoň většinu)
je celkem lehce rozšiřitelný

Cargo.toml

```
[dependencies]
diesel = { version = "1.4.4", features = ["postgres"] }
dotenv = "0.15.0"
```

CLI

nainstalujeme CLI

```
cargo install diesel_cli
```

a spustíme

```
cargo install diesel_cli --no-default-features --features postgres
```

pozn. je potřeba mít nainstalovaného klienta na práci s DB

Konfigurace připojení

Na DEV si uděláme connection string

```
echo DATABASE_URL=postgres://postgres:postgrespass@localhost/simple_chat > .env
```

Jen to nedávejte do GITu, ani to takto nedělejte pro produkční appky.

Setup Dieselu

```
diesel setup
```

vytvoří databázi, pokud neexistuje a složku na migrace

Migrace

verzují změny ve struktuře databáze a umožňují zmigrovat data mezi verzemi

```
diesel migration generate create_messages
```

Up.sql

```
CREATE TABLE messages (  
  id SERIAL PRIMARY KEY,  
  text TEXT NOT NULL,  
  created_at TIMESTAMP DEFAULT NOW()  
)
```

Down.sql

```
DROP TABLE messages
```

Spuštění migrace

Na produkci je potřeba dát spuštění migrace buď do nasazení nebo provést při startu aplikace.

```
diesel migration run
```

Připojení k DB

```
#[macro_use]
extern crate diesel;
extern crate dotenv;

use diesel::prelude::*;
use diesel::pg::PgConnection;
use dotenv::dotenv;
use std::env;

pub fn establish_connection() -> PgConnection {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");
    PgConnection::establish(&database_url)
        .expect(&format!("Error connecting to {}", database_url))
}
```

Struktura pro čtení

```
use diesel::types::Timestamp;

#[derive(Queryable)]
pub struct Message {
    pub id: i32,
    pub text: String,
    pub created_at: Timestamp,
}
```

Struktura pro zápis

```
use super::schema::messages;

#[derive(Insertable)]
#[table_name="messages"]
pub struct NewMessage<'a> {
    pub text: &'a str,
}
```

Insert

```
use self::simple_chat::*;
use self::models::*;
use self::diesel::prelude::*;

use self::models::{Message, NewMessage};

pub fn create_message<'a>(conn: &PgConnection, text: &'a str) -> Message {
    use schema::messages;

    let new_message = NewMessage {
        text: text,
    };

    diesel::insert_into(messages::table)
        .values(&new_message)
        .get_result(conn)
        .expect("Error saving new message")
}
```


Select

```
use self::simple_chat::*;
use self::models::*;
use self::diesel::prelude::*;

fn main() {
    use simple_chat::schema::messages::dsl::*;

    let connection = establish_connection();
    let results = messages.filter(published.eq(true))
        .limit(5)
        .load::(&connection)
        .expect("Error loading messages");

    let text = String::from("Prvni zprava");
    let message = create_message(&connection, text);

    println!("Displaying {} messages", results.len());
    for message in results {
        println!("{}", message.text);
        println!("-----\n");
    }
}
```



Jelikož všechno, co potřebujete si musíte dodat sami, tak musíte řešit i connection pooling.

Connection pooling umožňuje znovu používat databázové spojení.

r2d2

```
[dependencies]
```

```
diesel = { version = "1.0.0", features = ["postgres", "r2d2"] }
```

Connection pool

```
use diesel::pg::PgConnection;
use dotenv::dotenv;
use std::env;
use diesel::r2d2::{ Pool, PooledConnection, ConnectionManager, PoolError };

pub type PgPool = Pool<ConnectionManager<PgConnection>>;
pub type PgPooledConnection = PooledConnection<ConnectionManager<PgConnection>>;

fn init_pool(database_url: &str) -> Result<PgPool, PoolError> {
    let manager = ConnectionManager::<PgConnection>::new(database_url);
    Pool::builder().build(manager)
}

pub fn establish_connection() -> PgPool {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");

    PgConnection::establish(&database_url)
        .expect(&format!("Error connecting to {}", database_url))

    init_pool(&database_url).expect("Failed to create pool")
}
```

SQLx

SQLx je crate ke kontrole dotazu během kompilace. Nepoužívá žádný DSL.

Podporuje PostgreSQL, MySQL, SQLite, and MSSQL.

Podporuje různé asynchronní runtimey (async-std / tokio / actix) a TLS backendy (native-tls, rustls).

Cargo.toml

```
[dependencies]
# tokio + rustls
sqlx = { version = "0.5", features = [ "runtime-tokio-rustls" ] }
# async-std + native-tls
sqlx = { version = "0.5", features = [ "runtime-async-std-native-tls" ] }
```

```
use sqlx::mysql::MySQLPoolOptions;
use std::env;

#[tokio::main]
async fn main() -> Result<(), sqlx::Error> {
    let pool = MySQLPoolOptions::new()
        .max_connections(5)
        .connect(&env::var("DATABASE_URL")?.await?);

    let row: (i64,) = sqlx::query_as("SELECT ?")
        .bind(150_i64)
        .fetch_one(&pool).await?;

    assert_eq!(row.0, 150);

    Ok(())
}
```

```
use sqlx::postgres::PgPoolOptions;

#[async_std::main]
async fn main() -> Result<(), sqlx::Error> {
    let pool = PgPoolOptions::new()
        .max_connections(5)
        .connect("postgres://postgres:password@localhost/test").await?;

    let row: (i64,) = sqlx::query_as("SELECT $1")
        .bind(150_i64)
        .fetch_one(&pool).await?;

    assert_eq!(row.0, 150);

    Ok(())
}
```


Command

```
sqlx::query("DELETE FROM table").execute(&mut conn).await?;  
sqlx::query("DELETE FROM table").execute(&pool).await?;
```

Verifikování při kompilaci

```
let countries = sqlx::query!(
    "
    SELECT country, COUNT(*) as count
    FROM users
    GROUP BY country
    WHERE organization = ?
    ",
    organization
)
.fetch_all(&pool) // -> Vec<{ country: String, count: i64 }>
.await?;

// countries[0].country
// countries[0].count
```

query_as! do struktury

```
// no traits are needed
struct Country { country: String, count: i64 }

let countries = sqlx::query_as!(Country,
    "
SELECT country, COUNT(*) as count
FROM users
GROUP BY country
WHERE organization = ?
    ",
    organization
)
.fetch_all(&pool) // -> Vec<Country>
.await?;

// countries[0].country
// countries[0].count
```

Funkce pro práci s DB

```
async fn list_todos(pool: &SqlitePool) -> anyhow::Result<()> {
    let recs = sqlx::query!(
        r#"
SELECT id, description, done
FROM todos
ORDER BY id
"#
    )
    .fetch_all(pool)
    .await?;

    for rec in recs {
        println!(
            "- [{}] {}: {}",
            if rec.done { "x" } else { " " },
            rec.id,
            &rec.description,
        );
    }

    Ok(())
}
```

Repository pattern

```
#[async_trait]
pub trait TodoRepo {
    async fn add_todo(&self, description: String) -> anyhow::Result<i64>;
    async fn complete_todo(&self, id: i64) -> anyhow::Result<bool>;
    async fn list_todos(&self) -> anyhow::Result<()>;
}

struct PostgresTodoRepo {
    pg_pool: Arc<PgPool>,
}

impl PostgresTodoRepo {
    fn new(pg_pool: PgPool) -> Self {
        Self {
            pg_pool: Arc::new(pg_pool),
        }
    }
}
```

Repository pattern

```
#[async_trait]
impl TodoRepo for PostgresTodoRepo {
    async fn add_todo(&self, description: String) -> anyhow::Result<i64> {
        let rec = sqlx::query!(
            r#"
            INSERT INTO todos ( description )
            VALUES ( $1 )
            RETURNING id
            "#,
            description
        )
        .fetch_one(&self.pg_pool)
        .await?;

        Ok(rec.id)
    }

    async fn complete_todo(&self, id: i64) -> anyhow::Result<bool> {
        let rows_affected = sqlx::query!(
            r#"
            UPDATE todos
            SET done = TRUE
            WHERE id = $1
            "#,
            id
        )
        .execute(&self.pg_pool)
        .await?
        .rows_affected();

        Ok(rows_affected > 0)
    }
}
```

Migrations

```
use sqlx::migrate::Migrator;
use std::path::Path;

static EMBEDDED: Migrator = sqlx::migrate!("tests/migrate/migrations");

async fn main() -> anyhow::Result<()> {
    let runtime = Migrator::new(Path::new("tests/migrate/migrations")).await?;
}
```

Dotazy?

Děkuji za pozornost