



PV281: Programování v Rustu

Obsah

1. Docker a PlantUml
2. Přístupy pro práci s databází
3. Práce s proměnnými prostředí
4. SQLx
5. Práce s časem
6. Redis

Docker a PlantUML

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://docker.com/get-started>

Na Win 10/11 je nutné nainstalovat nejdříve `WSL2` nebo `Hyper-V`.

Alternativy k Dockeru

Vzhledem k licenční politice se dnes přechází od použití Dockeru pro kontejnerizaci.

Na produkci se používá v rámci Kubernetes `containerd` a trend je ho využít i pro lokální vývoj.

Další alternativy:

`podman`

`Rancher Desktop`

`colima` + `nerdctl`

Práce s Dockerem

Vyzkoušení:

```
docker run -d -p 80:80 docker/getting-started
```

Příkaz se připojí na *Docker Hub*, stáhne image, spustí démona a namapuje porty.

Docker compose

```
# Using the default 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

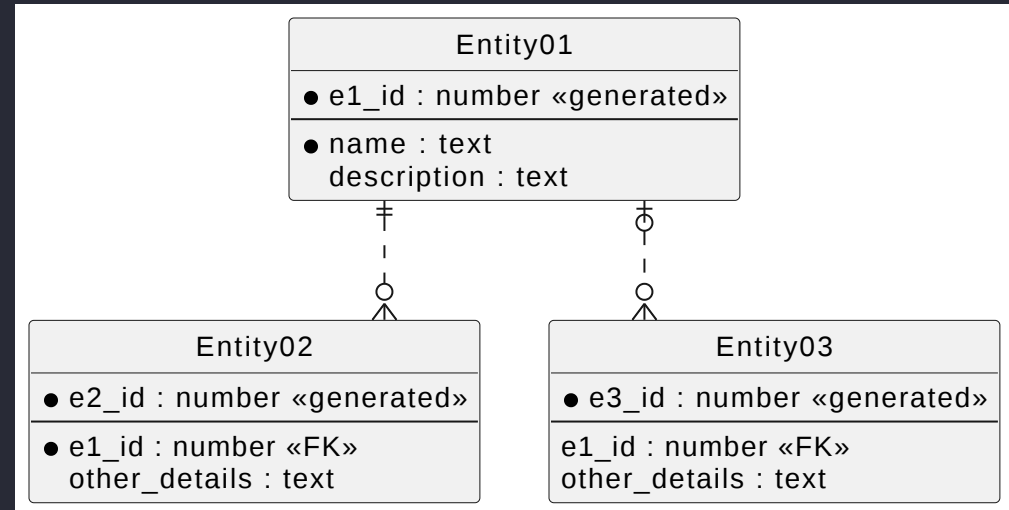
' avoid problems with angled crows feet
skinparam linetype ortho

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
```



Přístupy pro práci s databází

Postgres

Klasická relační SQL databáze.

Open-source a s velkým množstvím funkcí.

Velmi dobrý výkon i pro velké systémy

Občas se vyskytnou neřešené starší buggy.

Občas je výkon jiných DB systémů lepší.

Ručně vytvořené SQL dotazy

Výhody

Veškeré funkce jsou k dispozici.
Lehce lze optimalizovat výkon.

Nevýhody

Možnost SQL injection (při neznalosti).
Nutnost znát SQL.
Nutnost zavádět další jazyk do projektu.

Query Builder

Výhody

Většina funkcí je k dispozici.
Stále lze lehce optimalizovat výkon.
Nedochází k SQL injection.
Nezavádíme další jazyk do projektu.

Nevýhody

Nutnost znát SQL.
Nutnost navíc znát knihovnu, která SQL na pozadí vygeneruje.
Nemáme tolik možností jako při psaní čistého SQL.

ORM (Object–relational mapping)

Výhody

Omezuje množství možných útoků.
Jednoduchá a na vývoj rychlá práce s databází.
Vše typované, což pomáhá odhalit chyby.

Nevýhody

ORM knihovny často nepodporují všechny funkce.
Ztrácíme výkonnost – vygenerované dotazy nemusí být optimální.

Diesel

Výhody

Nejpoužívanější a jeden z nejrychlejších ORM v Rustu.
Eliminuje většinu runtime errorů při práci s databází.
Je celkem lehce rozšiřitelný.

Nevýhody

Na komplexnější dotazy si stejně musíte sami stavět dotaz.
Pro některé komplexnější věci je lepší využít jiné technologie.

SeaQL

Relativně nová (08/2021) rodina knihoven pro práci s DB:

`SeaQuery`, `SeaSchema`, `SeaORM`.

Výhodami oproti Dieselu mohou být async podpora, ucelený ekosystém, mockovatelnost.

Nevýhodami oproti Dieselu mohou být méně checků za kompilace, méně funkcionality.

Connection pooling

Vytváření a zavírání spojení je drahé a způsobuje latenci. Spojení si můžeme uložit a nechat jej otevřené, tím nemusíme platit za jeho nové vytvoření.

Díky poolu můžeme i ovlítnit minimální a maximální počet spojení.

Cachování dotazu

Databázové dotazy je vhodné cachovat.

Běžné je použití in-memory cache jako je Redis.

Vytáhnout výsledek z Redisu (pár klíč-hodnota) je levnější než zpracovat dotaz nad databází.

Práce s proměnnými prostředí

Práce s proměnnými prostředí

```
use std::env;

fn main() {
    let host_key = "HOST";
    let port_key = "PORT";
    let default_port = 8080;

    let host = match env::var(host_key) {
        Ok(val) => val,
        Err(err) => {
            println!("{}", err, host_key);
            process::exit(1);
        },
    };
}
```

Práce s proměnnými prostředí

Makra `env!` a `option_env!` se vyhodnocují za kompilace.

```
use std::env;

fn main() {
    let host_key = "HOST";
    let port_key = "PORT";
    let default_port = 8080;

    let host = env!(host_key);
    let port = option_env!(port_key);
}
```

crate envy

```
use serde::Deserialize;
use envy::envy;

#[derive(Deserialize, Debug)]
struct Config {
    foo: u16,
    bar: bool,
    baz: String,
    boom: Option<u64>
}

fn main() {
    match envy::from_env::<Config>() {
        Ok(config) => println!("{:#?}", config),
        Err(error) => panic!("{:#?}", error)
    }
}
```

crate

structopt

```
use structopt::StructOpt;

#[derive(StructOpt, Debug)]
#[structopt(name = "dbapp")]
struct Opt {
    /// Database URL
    #[structopt(env = "DATABASE_URL")]
    database_url: String,

    /// Port number
    #[structopt(env = "PORT")]
    port: Option<i32>,
}

fn main() {
    let opt = Opt::from_args();
    println!("{:#?}", opt);
}
```

Poznámka: `structopt` je dnes integrován v Clapu a není už vyvíjen.

Práce se souborem .env

Velmi používaná knihovna `dotenv`, která se objevuje v tutoriálech, už není dále udržovaná. Udržovaný fork je `dotenvy`.

```
// use dotenv::dotenv;
use dotenvy::dotenv;
use std::env;

fn main() {
    // dotenv().ok();
    dotenv().unwrap();

    for (key, value) in dotenvy::vars() {
        println!("{}", key, value);
    }
}
```


Konfigurace připojení

Pro vývoj si ukládáme connection string do `.env` souboru.
Tento soubor se neverzuje!

```
echo DATABASE_URL=postgres://postgres:postgrespass@localhost:5432/simple_chat > .env
#           ^scheme   ^user    ^password  ^hostname ^port^dbname
```

Načteme následně pomocí výše zmíněných metod.

SOILS

SQLx

SQLx je crate pro komunikaci s databází, poskytující kontrolu dotazu za kompilace.

Podporuje PostgreSQL, MySQL a SQLite.
Dříve do verze 0.7 i MSSQL.

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

Cargo.toml

[dependencies]

```
# tokio + rustls
```

```
sqlx = { version = "0.6", features = [ "postgres", "runtime-tokio-rustls", "offline" ] }
```

```
# async-std + native-tls
```

```
sqlx = { version = "0.6", features = [ "mysql", "runtime-async-std-native-tls", "offline" ] }
```

Migrace

SQLx poskytuje migrace skrze nástroj `sqlx-cli`.

Instalujeme přes Cargo:

```
cargo install sqlx-cli
```

Pro práci s migracemi je potřeba `.env` soubor s proměnnou `DATABASE_URL`.

Vytvoření databáze

```
sqlx database create  
sqlx database drop
```

Vytvoření migrace

Pro vytvoření migrace použijeme příkaz `migrate add`:

```
sqlx migrate add <name>  
Creating migrations/20211001154420_<name>.sql
```

Revertibilní migrace

Přepínačem `-r` vytvoříme revertibilní migraci:

```
sqlx migrate add -r user  
Creating migrations/20211001154420_user.up.sql  
Creating migrations/20211001154420_user.down.sql
```


Vytvoření tabulky v migraci

Soubor `user.up.sql`:

```
create table "user"  
(  
  user_id      uuid primary key default gen_random_uuid(),  
  username     text unique not null,  
  password_hash text          not null  
);
```

Soubor `user.down.sql`:

```
drop table user; --Nebezpečné!
```

Spuštění migrace

```
sqlx migrate run  
Applied migrations/20211001154420 user (32.517835ms)
```

Revert migrate

```
sqlx migrate revert  
Applied 20211001154420/revert user (32.517835ms)
```

Spuštění migrace v aplikaci

```
sqlx::migrate!("db/migrations") // <- Cesta ke složce s migracemi nebo ke konkrétnímu souboru migrace.  
  .run(&pool)  
  .await?;
```

Offline mode

Potřeba uložit metadata databáze do souboru `sqlx-data.json`:

```
cargo sqlx prepare
```

Poznámky

Aktuálnost metadat ověříme příkazem `cargo sqlx prepare --check`.

Proměnná prostředí `SQLX_OFFLINE=true` vynucuje offline kontrolu.

MySQL

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

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

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

    assert_eq!(row.0, 150);

    Ok(())
}
```

Postgres

```
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(())
}
```

Vytvoření spojení

Pokud potřebujeme jedno spojení:

```
use sqlx::Connection;  
  
let conn = SqliteConnection::connect("sqlite::memory:").await?;
```

Pokud budeme DB používat pravidelně nebo potřebujeme více spojení:

```
let pool = MySQLPool::connect("mysql://user:pass@host/database").await?;
```


Command

```
// Za použití jednoho připojení:  
sqlx::query("DELETE FROM table").execute(&mut conn).await?;  
  
// Za použití connection poolu:  
sqlx::query("DELETE FROM table").execute(&pool).await?;
```

Fetch

Vybíráme metodu podle toho, kolik řádků očekáváme.

Number of Rows	Method to Call	Returns	Notes
None	<code>.execute(...).await</code>	<code>sqlx::Result<DB::QueryResult></code>	For INSERT/UPDATE/DELETE without RETURNING. Only callable if the query returns no columns.
Zero or One	<code>.fetch_optional(...).await</code>	<code>sqlx::Result<Option<{adhoc struct}>></code>	Extra rows are ignored.
Exactly One	<code>.fetch_one(...).await</code>	<code>sqlx::Result<{adhoc struct}></code>	Errors if no rows were returned. Extra rows are ignored. Aggregate queries, use this.
At Least One	<code>.fetch(...)</code>	<code>impl Stream<Item = sqlx::Result<{adhoc struct}>></code>	Call <code>.try_next().await</code> to get each row result.
Multiple	<code>.fetch_all(...)</code>	<code>sqlx::Result<Vec<{adhoc struct}>></code>	

Jednoduchý SQL select

```
use anyhow::Result;
use dotenv::dotenv;
use std::env;
use sqlx::postgres::PgPoolOptions;

#[tokio::main]
async fn main() -> Result<> {
    dotenv::dotenv().unwrap();
    let pool = PgPoolOptions::new().connect(&env::var("DATABASE_URL")?).await?;

    let mut rows = sqlx::query("SELECT * FROM users WHERE email = $1")
        .bind(email)
        .fetch(&pool);

    while let Some(row) = rows.try_next().await? {
        let email: &str = row.try_get("email")?;
    }

    Ok(())
}
```

Map na dotazu

```
use anyhow::Result;
use dotenvy::dotenv;
use std::env;
use sqlx::postgres::PgPoolOptions;

#[tokio::main]
async fn main() -> Result<()> {
    dotenvy::dotenv().unwrap();
    let pool = PgPoolOptions::new().connect(&env::var("DATABASE_URL")?).await?;

    let mut stream = sqlx::query("SELECT * FROM users")
        .map(|row: PgRow| {
            // map the row into a user-defined domain type
        })
        .fetch(&pool);

    Ok(())
}
```

Přímé mapování do struktury

```
use anyhow::Result;
use dotenvy::dotenv;
use std::env;
use sqlx::postgres::PgPoolOptions;

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

#[tokio::main]
async fn main() -> Result<()> {
    dotenvy::dotenv().unwrap();
    let pool = PgPoolOptions::new().connect(&env::var("DATABASE_URL")?).await?;

    let mut stream = sqlx::query_as::<_, User>("SELECT * FROM users WHERE email = $1 OR name = $2")
        .bind(user_email)
        .bind(user_name)
        .fetch(&pool);

    Ok(())
}
```

Verifikace SQL při kompilaci

```
let countries = sqlx::query!( // <- Všimněte si, že jde o makro.  
    "  
        SELECT country, COUNT(*) as count  
        FROM users  
        GROUP BY country  
        WHERE organization = $1  
    ", // <- Pozor, tady musí být stringový literál a ne String.  
    organization  
)  
.fetch_all(&pool) // Návrátovým typem je `Vec<{ country: String, count: i64 }>`.  
.await?;  
  
// countries[0].country  
// countries[0].count
```

Makro `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 = $1
    ",
    organization
)
.fetch_all(&pool) // -> Vec<Country>
.await?;

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

Makro query_with!

```
use sqlx::{postgres::PgArguments, Arguments};
use std::env;

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

    let mut args = PgArguments::default();
    args.add(5);
    args.add("foo");

    sqlx::query_with!("insert into abc (a,b) values ($1, $2)", args);

    Ok(())
}
```


Vlastní 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(())
}
```

Transakce

```
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 mut tx = pool.begin().await?; // <- `begin` slouží i pro vytváření savepointu, pokud vnořujeme transakce.

    sqlx::query("INSERT INTO articles (slug) VALUES ('this-is-a-slug')")
        .execute(&mut tx).await?; // <- Otazník způsobí okamžitý rollback, pokud nastane chyba.

    tx.commit().await?;
    // tx.rollback().await?;

    Ok(())
} // <- Pokud nezavoláme commit, rollback je také provedený v rámci Drop na konci scope.
```

Repository pattern

Abstrahuje detaily práce s databází.

```
#[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)
    }
}

// Continued on the next half...
```

```
// ...continued from the previous half.

    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)
    }
}
```

Práce s časem

Nejpoužívanější jsou knihovny `time` a `chrono`,
příčemž `chrono` je postavené nad `time`.

Dlouhou dobu `chrono` zůstávalo závislé na staré verzi `time` 0.1.

Obě knihovny v minulosti neměli na určitou dobu maintainera,
proto uživatelé přecházeli mezi nimi.

Dnes jsou obě udržované a také vzájemně nekompatibilní.

`chrono` je používanější a obsáhlejší.

`time` je zase mnohem jednodušší na pochopení.

`SQLx` v případě povolení obou preferuje použití `time`.

time

[dependencies]

```
time = { version = "0.3", features = [ "macros" ] }  
sqlx = { version = "0.5", features = [ "runtime-tokio-rustls", "time" ] }
```

time

```
use time::{Date, PrimitiveDateTime, OffsetDateTime, UtcOffset};
use time::Weekday::Wednesday;

fn main() {
    let date = Date::from_iso_week_date(2022, 1, Wednesday).unwrap();
    let datetime = date.with_hms(13, 0, 55).unwrap();
    let datetime_off = datetime.assume_offset(UtcOffset::from_hms(1, 2, 3).unwrap());

    println!("{date}, {datetime}, {datetime_off}");
    // 2022-01-01, 2022-01-01 13:00:55.0, 2022-01-01 13:00:55.0 +01:02:03
}
```

time - makra

```
use time::macros::{date, datetime};

fn main() {
    let date = date!(2022-01-01);
    let datetime = datetime!(2022-01-01 13:00:55);
    let datetime_off = datetime!(2022-01-01 13:00:55 +1:02:03);

    println!("{date}, {datetime}, {datetime_off}");
    // 2022-01-01, 2022-01-01 13:00:55.0, 2022-01-01 13:00:55.0 +01:02:03
}
```


time - offset

```
fn main() {
    assert_eq!(
        datetime!(2000-01-01 0:00 UTC).to_offset(offset!(-1)).year(),
        1999,
    );

    let sydney = datetime!(2000-01-01 0:00 +11);
    let new_york = sydney.to_offset(offset!(-5));
    let los_angeles = sydney.to_offset(offset!(-8));

    assert_eq!(sydney.hour(), 0);
    assert_eq!(sydney.day(), 1);
    assert_eq!(new_york.hour(), 8);
    assert_eq!(new_york.day(), 31);
    assert_eq!(los_angeles.hour(), 5);
    assert_eq!(los_angeles.day(), 31);
}
```

Redis

Redis

```
# if you use tokio
redis = { version = "0.22.1", features = ["tokio-native-tls-comp"] }

# if you use async-std
redis = { version = "0.22.1", features = ["async-std-tls-comp"] }
```

Základní operace

```
use redis::Commands;

fn fetch_an_integer() -> redis::RedisResult<isize> {
    let client = redis::Client::open("redis://127.0.0.1/");
    let mut con = client.get_connection()?;

    let _ : () = con.set("my_key", 42)?;

    con.get("my_key")
}
```

Poznámky k implementaci

`redis-rs` nemá connection pool. Pro jeho vytvoření je možné použít `bb8` nebo `deadpool`.

Ukázka bb8 implementace

```
use futures_util::future::join_all;
use bb8_redis::{ bb8, redis::{cmd, AsyncCommands}, RedisConnectionManager };

#[tokio::main]
async fn main() {
    let manager = RedisConnectionManager::new("redis://localhost").unwrap();
    let pool = bb8::Pool::builder().build(manager).await.unwrap();
    let mut handles = vec![];

    for _i in 0..10 {
        let pool = pool.clone();

        handles.push(tokio::spawn(async move {
            let mut conn = pool.get().await.unwrap();
            let reply: String = cmd("PING").query_async(&mut *conn).await.unwrap();

            assert_eq!("PONG", reply);
        }));
    }

    join_all(handles).await;
}
```

Dotazky?

Děkuji za pozornost