Frameworks

A collection of tutorials for some frameworks I often use.

- SQLx
- Actix-Web
- Tracing
- Serde
- config
- SeaORM

SQLx

SQLx is a SQL framework for Rust, which allows to write SQL queries and check them on compile time. It also comes with a migration logic to update your database.

Install SQLx

To install the SQLx, run the following command in the terminal

```
cargo install sqlx-cli
```

In your Rust project, install sqlx using the following command

```
cargo add sqlx --features postgres,runtime-tokio

# Instead of postgres, you can also use the following databases:
# postgres, mysql, mssql, sqlite

# Also instead of runtime-tokio, you can use:
# runtime-tokio, runtime-async-std
# For actix-web projects, use runtime-tokio
```

Database Migration

To create a migration, you need to set the environment variable `DATABASE_URL`, then execute the following command to create the database:

```
# Create the database
sqlx database create
```

To create a migration step, execute the following command. Replace "name" with a description in snake case, which describes the migration step.

```
# Add a migration, replace name
sqlx migrate add -r name

# Or using sequential numbers instead of timestamps
sql migrate add -r -s name
```

The flag `-r` will create two migration files, one with the pattern `timestamp_name.up.sql` and one with the name `timestamp_name.down.sql` - the one is the migration and the second is the revert script. Without it, it will only create a `timestamp_name.sql` file. You can also use the flag `-s`, then instead of a timestamp a sequence will be used (`0001_name.up.sql`).

```
# Execute migration
sqlx migrate run
```

Migrate per code

You can embed the migrations into the application, so that the migration will be executed on startup:

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

let pool = PgPoolOptions::new()
        .max_connections(5)
        .connect(&database_url)
        .await
        .expect("Failed to connect to database");

sqlx::migrate!()
        .run(&pool)
        .await
        .expect("Failed to run migrations");
```

Actix-Web

actix-web is a framework for writting web applications and REST APIs.

Installation

```
# To install:
cargo add actix-web

# Recommended for validation
cargo add validator actix-web-validator
```

Simple example

```
#[get("/hello")
async fn hello() -> Response {
  HttpResponse::Ok().body("Hello!")
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(move || {
        App::new()
            .app_data(here_you_can_add_dependencies)
            .service(hello)
    })
    .bind(("0.0.0.0", 8080))?
    .run()
    .await
}
```

JSON request bodies

To parse a JSON body from a request, you define a struct, which represents the body, and then use the `web::Json` extractor.

```
#[derive(Deserialize)]
struct HelloRequest {
  name: String,
}

#[post("/hello")
async fn hello(payload: web::Json<HelloRequest>) -> Response {
  let name = payload.name;
  HttpResponse::Ok().body(format!("Hello, {name}!"))
}
```

You can also use the actix-web-validator package to create a validation. The request must then match your rules:

```
#[derive(Deserialize, Validate)]
struct HelloRequest {
  #[validate(length(min = 1, max  = 32)]
  name: String,
}

#[post("/hello")
async fn hello(payload: actix_web_validator::Json<HelloRequest>) -> Response {
  let name = payload.name;
  HttpResponse::Ok().body(format!("Hello, {name}!"))
}
```

Scopes

You can group multiple services into a scope, to give them the same prefix.

```
let v1 = web::scope("/api/v1")
  .service(list_entities)
  .service(create_entity)
  .service(delete_entity);
```

```
App::new()
  .service(v1)
```

Custom extractors

Extractors are used in the parameter lists of your service functions. You can write own to **extract information from a request** (e.g. get a user by authorization header). To write such a custom header, you need to create a struct (which will be used in the parameter list and contain the extracted information) and then implmenet the FromRequest trait:

```
struct MyExtractor {
  ...
}

impl FromRequest for MyExtractor {
    type Error = actix_web::Error;
    type Future = future_utils::LocalBoxFuture<'static, Result<Self, Self::Error>>;

    fn from_request(req: &HttpRequest, payload: &mut Payload) -> Self::Future {
        let req = req.clone();

        Box::pin(async move {
            // Do extraction...

            Ok(MyExtractor { ... })
        })
    }
}
```

To access app_data, you can use the request:

```
let pool = match req.app_data::<web::Data<Pool<Postgres>>>() {
    Some(pool) => pool,
    None => panic!(),
};
```

Tracing

Tracing is a library for logging. It consists of multiple libraries. You need at least `tracing`, which is the base of the logging system, and `tracing_subscriber`, which defines the subscriber of the tracing messages.

```
# To install
cargo add tracing tracing_subscriber

# I recommend the use of the env filter
cargo add tracing tracing_subscriber --features env-filter
```

Setup

First you need to setup how the tracing messages are subscribed/used. You can setup a basic logging using the following code:

```
tracing_subscriber::fmt()
  .with_target(false)
  .with_level(true)
  .compact()
  .init();

# Or optional with the default env filter
tracing_subscriber::fmt()
  .with_target(false)
  .with_level(true)
  .with_env_filter(EnvFilter::from_default_env())
  .compact()
  .init();
```

Serde

Serde (**Ser**ialize and **De**serialize) is a library to serialize and deserialize data to or from a struct. This is usually used with language based crates like **serde_json** for JSON or **serde_yaml** for YAML.

To use it, install it, and then you can use the derives on your structs.

```
cargo add serde --features derives
cargo add serde_json
```

```
#[derive(Serialize, Deserialize)]
struct MyData {
  name: String,
  #[serde(rename = "invoiceAddress")]
  invoice_address: InvoiceAddress,
  #[serde(skip)]
  do_not_serialize: bool,
}
```

config

config (crates.io) is a library for building configuration structs from sources like env variables, yaml files and so on. You can combine it with validator to build a full configuration for your application.

Install

```
# Install config
cargo add config
cargo add serde --features derive

# Optional libraries I recommend
cargo add dotenv
cargo add validator --features derive
```

Example

In this example, I have created a struct, which holds `base_url` and `port`. `base_url` is mandatory while `port` has a default value. I can load it using a config file, or using environment variables.

I also included `dotenv`, so the environment variables are optionally read using an `.env` file.

```
use config::{Environment, File};
use dotenv::dotenv;
use serde::Deserialize;
use validator::Validate;

use crate::errors::AppError;

#[derive(Debug, Clone, Deserialize, Validate)]
pub struct Config {
    #[validate(length(min = 1))]
    pub base_url: String,
    #[validate(range(min = 1, max = 65535))]
```

```
    pub port: u16,
}

pub fn load() -> Result<Config, AppError> {
    dotenv().ok();

    let settings = config::Config::builder()
        .set_default("port", 8080)?
        .add_source(File::with_name("config").required(false))
        .add_source(Environment::default())
        .build()?;

    let config: Config = settings.try_deserialize()?;
    config
        .validate()
        .map_err(|err| AppError::ConfigValidation(err))?;

    Ok(config)
}
```

SeaORM

SeaORM (from SeaQL) is an ORM for Rust.

Installation

When Installing SeaORM, you need to define what Runtime to use and what database driver to use.

TODO

Install the CLI tool

TODO

Initialize migration

To add migration, TODO: create initial migration, setup Cargo.toml

Write migrations

TODO

Execute migrations

TODO

Use SeaORM

TODO