Rust

Basics

The basics of Rust.

Smart Pointers

Smart Pointers are references to data, so you can use it, without taking ownership of this data. But before talking about Smart Pointers, lets take a look at normal references.

Normal references

You can create references to variables. This gives code the ability to work with those variables, without taking the ownership.

Reference

TODO

Mutable Reference

TODO

Smart Pointers

Smart Pointers are often required, because they give you additional functionalities, where normal references do not fit the problem. They behave like a pointer, but add safety mechanisms to prevent wrong memory management.

Box

Variables are usually stored in the stack. But sometimes you need to allocate data on the heap memory. A good example is, when the size of the data struct can not be calculated on compile time. The Box smart pointer is basically a pointer to the struct on the heap.

```
// 5 is allocated in the heap,
// and "Pointer" is a smart pointer
// pointing on "5"
let pointer = Box::new(5);

// To use the value, dereference the pointer
println!("value = {}", *pointer);

// Implicit derefrencing may also work
```

```
// because Box implements the Deref trait
println!("value = {}", pointer);
```

Rc

The ownership of data is always bould to one place. One variable has the responsibility to clean up the memory used when done. but at some points mutiple places need to have ownership of those data.

Rc is a smart pointer with a **reference counter**, which allows to share ownership on multiple places. Each time another place needs the memory, the Rc pointer is copied and increases the reference counter. And every time the life time of a Rc ends, it decreases the reference counter. Eventually the counter becomes zero and the stored data will be decallocated.

Rc is used to share ownership in **single threaded** code and provides **read only access** to the data.

```
// create a Rc pointer
let magic_number = Rc::new(1337);

{
  // clone the pointer to create multiple references
  // and allow it to be used somewhere
  let magic_clone = Rc::clone(&magic_number);
  // Now the counter is at 2
}
// The lifetime of magic_clone ends and the
// counter is decreased to 1 again

println!("magic number = {}", *magic_number);
```

Arc

TODO

RefCell

TODO

Cell

TODO

Mutex

TODO

RwLock

TODO

Cow

A Cow pointer (Copy-on-read) points to borrowed read only data. But it allows to create a mutable, by cloning those data and creating a mutable pointer.

```
# Example from Rust docs

use std::borrow::Cow;

fn abs_all(input: &mut Cow<'_, [i32]>) {
    for i in 0..input.len() {
        let v = input[i];
        if v < 0 {
            // Clones into a vector if not already owned.
            input.to_mut()[i] = -v;
        }
    }
}

// No clone occurs because `input` doesn't need to be mutated.
let slice = [0, 1, 2];
let mut input = Cow::from(&slice[..]);
abs_all(&mut input);

// Clone occurs because `input` needs to be mutated.
let slice = [-1, 0, 1];
let mut input = Cow::from(&slice[..]);
abs_all(&mut input);

// No clone occurs because `input` is already owned.
let mut input = Cow::from(vec![-1, 0, 1]);
abs_all(&mut input);
```

Here, `input` initially points to `slice`, hence doesn't require as much space as a copy. For read-only operations it is efficient. But when `slice` contains a negative value which needs to be modified, `Cow` creates a copy when calling `to_mut`, which then can be safely modified.

This kind of pointer is useful, when you have larger data, which you want to use without modifying it, but you are **not sure whether you need to modify it or not**. When you don't need to modify the data, it saves space by borrowing the data instead.

Weak

TODO

Pin

TODO

Macros

An overview and examples for the diverse types of macros you can use in Rust.

Declarative Macros

Declarative macros look like a function: e.g. `println!()`, `format!()` or `todo!()`. They have an exclamation mark between the name and the parameter list.

Declarative macros are like "shortcuts", it is usually more code, packed in a code block, which will generated at the place where it is called. This has the benefit, that it is not a function call, hence does not add to the stack or needs to consider moving. It is like a template, which will be copied in place.

Procedural Macros

Procedural macros are "headers" added above e.g. a struct, **take those information and generate different code out of it**. They look like this: `#[example]`. They are way more complex, but give you the **ability to generate complex code**.

Procedural Macros takes an object and **produces a stream of tokens**, a "TokenStream". You then manipulate the token stream, so the code at this place will be the desired generated result.

Those generated results are not "copied in the source", they are cached. You only see your source code and the macro call. But there is a command to show, how the generated result looks like. But it resolves not only your macro, it resolves all macros, hence debugging using it can be helpful, but hard work.

Derive Macros

```
// Example

#[derive(Debug, Clone)]
struct Example {
  a: i32,
  b: i32,
}
```

Attribute-Like Macros

Attribute-Like macros modify **structs, functions or modules**.

```
//example

use my_macros::log_call;

#[log_call]
fn add(a: i32, b: i32) -> i32 {
```

```
    a + b
}
```

Function-Like Macros

Function-Like macros automatically **generate a function**.

```
// Example


create_hello_fn!(say_hello);


fn main() {
    say_hello();
}
```

They **look similar to declarative macros**, but they don't use a template, they use a TokenStream and you are able to analyse the syntax and generate the code (by using *syn and quote*).

Frameworks

A collection of tutorials for some frameworks I often use.

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");
```

Frameworks

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,
}
```

Frameworks

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

Tricks

Allow everything what can be a str reference

```
fn do_something<S: AsRef<str>>(input: S) {
  let input_str = input.as_ref();
  println!("This is a &str: {}", input_str);
}

do_something("Hello World");
do_something("Hello World".to_string());
```

Cross compilation

You can cross compile your Rust code using the target specifier. For example:

```
cargo build --target x86_64-unknown-linux-gnu --release
```

You can list all targets using `rustup target list` and add the targets you need. But often you need an environment with the correct tooling. This is made easier using the **cross** (Github) tool.

Install cross

You can simply use cargo to install cross:

```
cargo install cross --git https://github.com/cross-rs/cross
```

Cross requires either **docker or podman** on your machine, because it uses docker images to provide the necessary tooling.

It automatically chooses the engine, but you can also set it using the environment variable `CROSS_CONTAINER_ENGINE`.

Build using cross

You can now build using cross by replacing the "cargo build" with "cross build":

```
# cross build --target <your target> <additional parameters>
cross build --target x86_64-unknown-linux-gnu --release

# Or with a different container engine
CROSS_CONTAINER_ENGINE=podman cross build --target x86_64-unknown-linux-gnu --release
```