Tips for writing a web service in Rust

Apoorv Kothari
toidiu.com
@toidiuCodes

content

- db management
- testing
- code hardening
- logging
- auth
- warp framework
- coding patterns

https://github.com/toidiu/fin-public

db management

- migrations with `diesel_cli`
- ORM vs raw SQL
- postgres crate
- postgres-mapper
- db testing

db management (cont..)

migrations with diesel_cli

- `diesel setup`
- `diesel migration generate`
- `diesel migration run`
- `diesel migration revert`

```
▾ migrations/
  ▾ 00000000000000_diesel_initial_setup/
      down.sql
      up.sql
  ▾ 2018-10-07-022941_init/
      down.sql
      up.sql
  ▾ 2018-10-07-232226_fake_data/
      down.sql
      up.sql
```

```
➜ ~/project/fin/service git:(master) export DATABASE_URL='postgres://localhost/r_fin';
➜ ~/project/fin/service git:(master) diesel migration list
Migrations:
  [X] 00000000000000_diesel_initial_setup
  [X] 2018-10-07-022941_init
  [X] 2018-10-07-232226_fake_data
```

db management (cont..)

ORM vs raw SQL

ORM (Diesel)

- con: can be difficult to write complex queries and need to learn a new framework
- pro: queries typed checked at compile time!

raw SQL

- pro: simply write the SQL you want
- con: need to write queries manually which can be error prone

db management (cont..)

postgres crate

- flexibility of raw SQL

  but

- error prone:
  - if a new field is added
  - if we rename the table

```
let rows = &self
    .conn
    .query(
        "SELECT fk_port_g_id, fk_tic_id, goal_per, ord FROM tic_goal
        WHERE fk_port_g_id = $1",
        &[port_g_id],
    ).map_err(|err| FinError::DatabaseErr(err.to_string()))?;

let ret = rows
    .iter()
    .map(|row| db_types::TickerGoalData {
        fk_port_g_id: row.get(0),
        fk_tic_id: row.get(1),
        goal_per: row.get(2),
        ord: row.get(3),
    }).collect::<Vec<db_types::TickerGoalData>>();
```

db management (cont..)

postgres crate

- ideal SQL
  - auto retrieve field names and table name
  - automatically extract results in order

```
let stmt = &format!(
    "SELECT {} FROM {} WHERE email = $1",
    &db_types::UserData::sql_fields(),
    &db_types::UserData::sql_table(),
);

let rows = &self.conn.query(stmt, &[&email]).map_err(|err| {
    error!(self.logger, "{}: {}", line!(), err);
    FinError::DatabaseErr
})?;

let ret: ResultFin<db_types::UserData> = rows
    .iter()
    .next()
    .map(|row| {
        db_types::UserData::from_postgres_row(row).map_err(|err| {
            error!(self.logger, "{}: {}", line!(), err);
            FinError::DatabaseErr
        })
    })
    .ok_or(FinError::DatabaseErr)?;
```

db management (cont..)

postgres-mapper

- derive procedural macro
  - UserData::from_postgres_row(row) -> **Result<UserData, _>**
- attribute procedural macro
  - UserData::sql_fields() -> users.id, users.email
  - UserData::sql_table() -> users

```rust
#[derive(PostgresMapper)]
#[pg_mapper(table = "users")]
pub struct UserData {
    pub id: i64,
    pub email: String,
}
```

```toml
[dependencies.postgres-mapper]
features = ["postgres-support"]
git = "https://github.com/toidiu/postgres-mapper"

[dependencies.postgres-mapper-derive]
features = ["postgres-mapper", "postgres-support"]
git = "https://github.com/toidiu/postgres-mapper"
```

db management (cont..)

postgres-mapper

- derive procedural macro
  - UserData::from_postgres_row(row) -> Result<UserData, _>
- attribute procedural macro
  - UserData::sql_fields() -> **users.id, users.email**
  - UserData::sql_table() -> **users**

```rust
#[derive(PostgresMapper)]
#[pg_mapper(table = "users")]
pub struct UserData {
    pub id: i64,
    pub email: String,
}
```

```toml
[dependencies.postgres-mapper]
features = ["postgres-support"]
git = "https://github.com/toidiu/postgres-mapper"

[dependencies.postgres-mapper-derive]
features = ["postgres-mapper", "postgres-support"]
git = "https://github.com/toidiu/postgres-mapper"
```

db management(cont..)

db testing

- setup
- run test
- teardown

```rust
pub fn run_test<T>(test: T) -> ()
where
    T: FnOnce(&str) -> () + std::panic::UnwindSafe,
{
    let db_name = Self::get_test_db_name();

    Self::setup(&db_name);
    let result = std::panic::catch_unwind(|| test(&db_name));
    Self::teardown(&db_name);

    assert!(result.is_ok())
}
```

```rust
#[test]
fn test_get_user() {
    TestHelper::run_test(|db_name| {
        let db = TestHelper::get_test_db(db_name);
        let res = db.get_user("apoorv@toidiu.com");
        assert_eq!(res.is_ok(), true);
        assert_eq!(res.unwrap().email, "apoorv@toidiu.com");
    })
}
```

db management(cont..)

db testing

- setup DB for testing
  - get a connection (real postgres instance)
  - re-use migration scripts!!

```rust
fn setup(db_name: &str) {
    // create database
    let db_conn = Connection::connect(CLUSTER_URI, TlsMode::None)
        .expect("unable to create db conn");
    db_conn
        .execute(&format!("CREATE DATABASE {name};", name = db_name), &[])
        .expect("unable to create db");

    // apply schema and add fake data
    let c_str = format!("{}/{}", CLUSTER_URI, db_name);
    let conn = Connection::connect(
        Self::get_test_db_uri(db_name).as_str(),
        TlsMode::None,
    )
    .unwrap();
    let init =
        fs::read_to_string("migrations/2018-10-07-022941_init/up.sql")
            .expect("file not found");
    let fake_data =
        fs::read_to_string("migrations/2018-10-07-232226_fake_data/up.sql")
            .expect("file not found");

    conn.batch_execute(&init).unwrap();
    conn.batch_execute(&fake_data).unwrap();
}
```

code hardening

- error handling
- new types

code hardening(cont..)

error handling

- declare custom App Error type and type alias for custom App Result **(use failure crate)**

```
pub enum FinError {
    NotLoggedIn,   // user is not logged in
    ServerErr,     // internal server error
    BadRequestErr, // a request is malformed (form has bad data)
    NotFoundErr,   // resource not found
    DatabaseErr,   // any database related error
}
```

```
pub type ResultFin<T> = Result<T, FinError>;
```

- all functions should only return ResultFin
  - cast errors to custom error

code hardening(cont..)

newtypes

- new types: *"newtypes are a zero-cost abstraction: they introduce a new, distinct name for an existing type, with* ***no runtime overhead*** *when converting between the two types."*
  - restrict creation to macro
  - restrict access to well-names fn
  - do NOT impl trait `Deref`

```
pub struct TickerId(i64);
```

```
macro_rules! tic_id(
    ($s:expr) => (
        TickerId::new($s)
    )
);
```

```
pub fn get_ticker_id(&self) -> &i64 {
    &self.0
}
```

logging

- slog
- line!

logging(cont..)

slog compassable

- composable: logging plugins (async, formatting, file vs network storage)

```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

let fuse = slog_async::Async::new(formatter.fuse()).build().fuse();
slog::Logger::root(fuse, o!("crate" => "fin", "version" => env!("CARGO_PKG_VERSION")))
```

logging(cont..)

slog structured

- structured: data should be machine searchable (think JSON)

```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

let fuse = slog_async::Async::new(formatter.fuse()).build().fuse();
slog::Logger::root(fuse, o!("crate" => "fin", "version" => env!("CARGO_PKG_VERSION")))
```

logging(cont..)

slog contextual

- contextual: trace code path via tags

```
logger.new(o!("mod" => "data"))
```

logging(cont..)

line!

- `lineError!` macro to get line info with your logging
  - works because macro expands to rust code at compile time

```
macro_rules! lineError(
    ($logger:expr, $msg:expr) => (
        error!($logger, "line: {} - {}", line!(), $msg);
    )
);
```

```
lineError!(
    self.logger,
    format!("{}. user_id: {:?}", err, &user_id)
);
```

auth

- password management: libpasta
- stateless user session: paseto

auth(cont..)

password management

- **libpasta** <https://libpasta.github.io/>
  - Easy-to-use password storage with strong defaults.
    - `` `libpasta::hash_password(&password);` ``
    - `` `libpasta::verify_password(&user.password_hash, &password)` ``
  - Tools to provide parameter tuning for different use cases.
  - Automatic migration of passwords to new algorithms.
    - `` `algo2 (algo1 ( password ))` ``

auth(cont..)

a small digression to discuss security

- session database: the token is a random session value which is stored in a database
  - the token can be invalidated, rotated, can link to a user's info and lives in a single place which is the database
  - this form of auth is harder to get wrong, speaking cryptographically
- stateless user session: uses a stateless token to authenticate the user
  - has more gotchas. need to think about expiry, key rotation, maybe versioning, which values to include...
- **Disclaimer**: cryptography is difficult! You should get an expert if it really matters to you

auth(cont..)

stateless session token

- stateless user session: **paseto**
  - has more gotchas. need to think about expiry, key rotation, maybe versioning, which values to include...
  - Paseto is everything you love about JOSE (JWT, JWE, JWS) without any of the many design deficits that plague the JOSE standards.
  - It's incredibly unlikely that you'll be able to use Paseto in an insecure way.

warp framework

a composabe web framework

<https://github.com/seanmonstar/warp>

- the good
- the bad

warp framework(cont..)

the good

- composable!
- `warp::Filter` will eventually implement `tower::Service`!
- leverages Rust traits and built on hyper
    - no macro magic / no actor framework

```
// AUTH
let with_auth = warp::cookie::optional(&auth::SESS_COOKIE_NAME).and_then(
    |opt_sess: Option<String>| match opt_sess {
        Some(sess) => auth::parse_sess(&sess)
            .map_err(|err| warp::reject::custom(FinError::NotLoggedIn)),
        None => Err(warp::reject::custom(FinError::NotLoggedIn)),
    },
);
```

```
// POST -> portfolio/actual
let create_port_a = warp::post2()
    .and(portfolio_path)
    .and(warp::path("actual"))
    .and(warp::path::end())
    .and(with_auth)
    .and(warp::body::json())
    .and(with_portfolio_backend)
    .and_then(portfolio_server::create_port_a);
```

```
fn create_port_a(
    &self,
    user_id: &server::UserId,
    goal_id: &i64,
    stock_percent: &f32,
) -> ResultFin<server::PortfolioActualResp>;
```

warp framework(cont..)

the bad

- trait magic

  <https://github.com/seanmonstar/warp/blob/v0.1.16/src/generic.rs>

- bad type error reporting

  <https://pastebin.com/raw/GpFvVQW6>

coding patterns

- state-functional pattern
- mod re-exports
- pub identifiers
- type per app interface

coding patterns(cont..)

state-functional pattern

- functional + performance (obj creation)
- 1 initial object per request
- mutated the state across request

```rust
pub fn new(port: PortfolioState) -> Self {
    let initial_actual_tickers = port.get_actual_tickers().clone();
    BuyNext {
        init_state: port,
        evolved_actual: initial_actual_tickers,
        actions: Vec::new(),
        buy_value: 0.0, // default uninitialized value
        action_summary: HashMap::new(),
    }
}
```

```rust
pub fn get_next_action(
    buy_next: &mut BuyNext,
    buy_amount: f64,
    port_state: PortfolioState,
) -> Option<Action> {
```

coding patterns(cont..)

mod re-exports

- re-export all pub interface.. and have single point of entry

- re-exports from base mod

```rust
// portfolio/mod.rs

mod actual;
mod state;

pub use self::{
    actual::{PortfolioActual, TickerActual},
    state::PortfolioState,
};
```

- exposed pub interface from the base mod `*portfolio::*`

```rust
portfolio::TickerActual :new(1, 1, 1, 1, 0.0);
```

coding patterns(cont..)

pub identifiers

- add identifier to all `` `pub(self, super,crate, path) fn` ``
  - `` `self` `` expose for current mod
  - `` `super` `` expose for parent mod
  - `` `crate` `` expose for current crate
  - `` `path` `` exposes for path

    (generally prefer the others because they are relative)

coding patterns(cont..)

type per app interface

- restrict change to a smaller surface area (server, internal, db)

```rust
// server/api.rs

#[derive(Serialize, Deserialize, Debug)]
pub struct PortfolioActualResp {
    pub id: i64,
    pub fk_user_id: i64,
    pub fk_port_g_id: i64,
    pub stock_percent: f32,
    pub deviation: f32,
    pub version: i32,
    pub last_updated: DateTime<Utc>,
    pub tickers_actual: Vec<portfolio::TickerActual>,
}
```

```rust
// portfolio/actual.rs

#[derive(Serialize, Deserialize, Clone, Debug)]
pub struct PortfolioActual {
    pub id: i64,
    pub fk_user_id: i64,
    pub fk_port_g_id: i64,
    pub stock_percent: f32,
    pub deviation_percent: f32,
    pub version: i32,
    pub last_updated: DateTime<Utc>,
    pub tickers_actual: HashMap<TickerId, TickerActual>,
}
```

```rust
// data/db_types.rs

#[derive(Debug, PostgresMapper)]
#[pg_mapper(table = "actual_port")]
pub struct ActualPortData {
    pub id: i64,
    pub fk_user_id: i64,
    pub fk_port_g_id: i64,
    pub stock_percent: f32,
    pub deviation: f32,
    pub version: i32,
    pub last_updated: DateTime<Utc>,
}
```

Fin

Apoorv Kothari
toidiu.com
@toidiuCodes