Tips for writing a web server and beyond

Apoorv Kothari
toidiu.com
@toidiuCodes

content

- designing with lifetimes
- db management
- logging
- code hardening

working code available at:

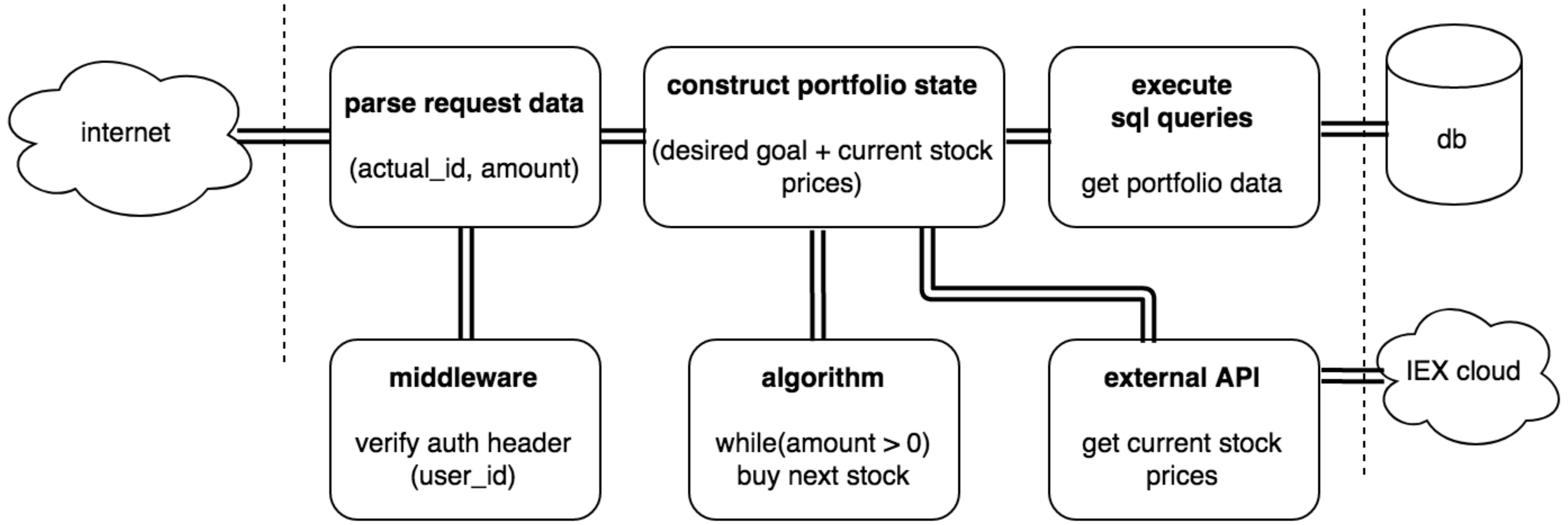
github.com/toidiu/fin-public

designing with lifetimes

designing with lifetimes(cont..)

system boundaries

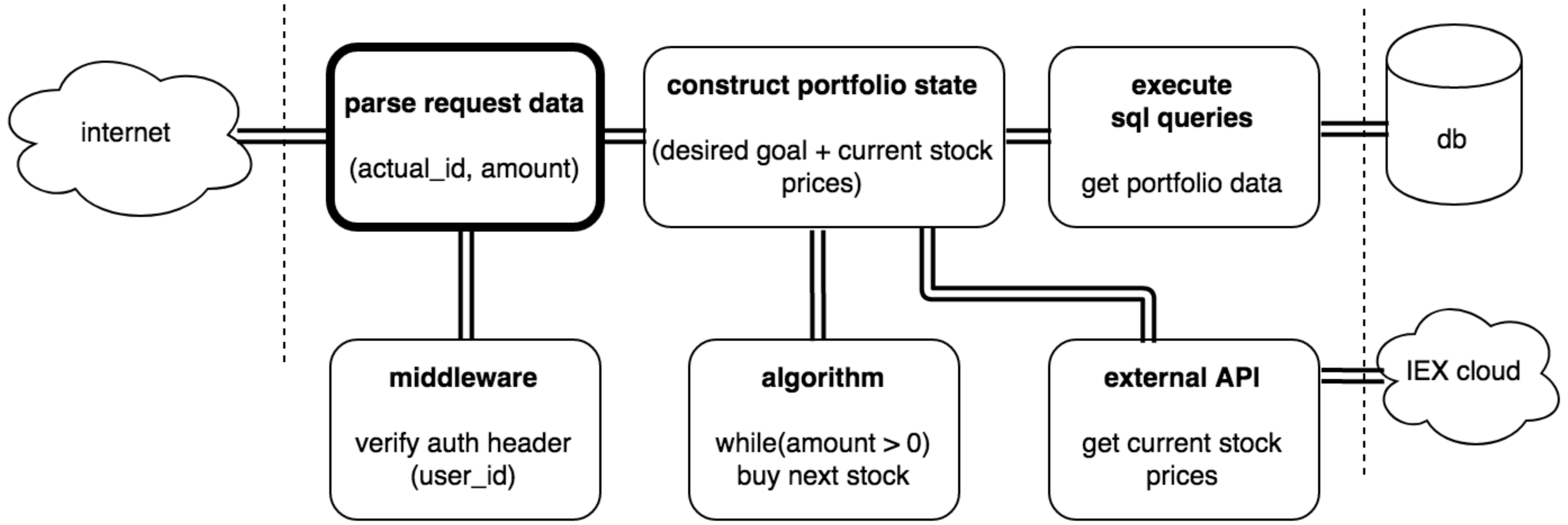
GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

system boundaries

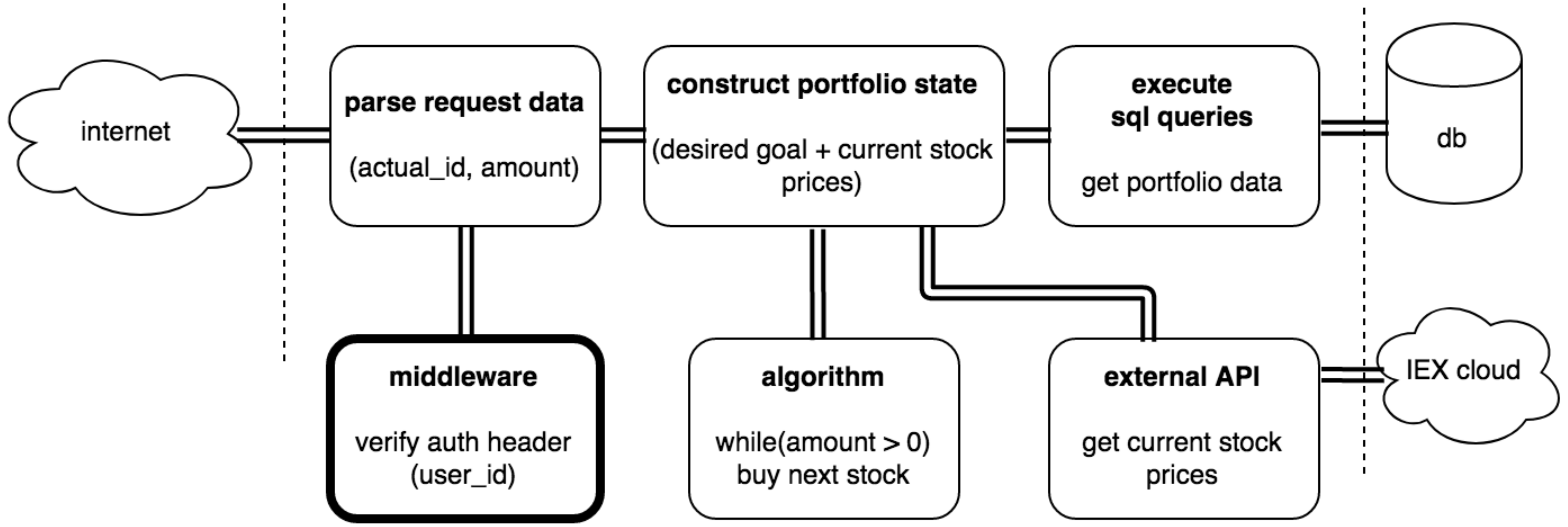
GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

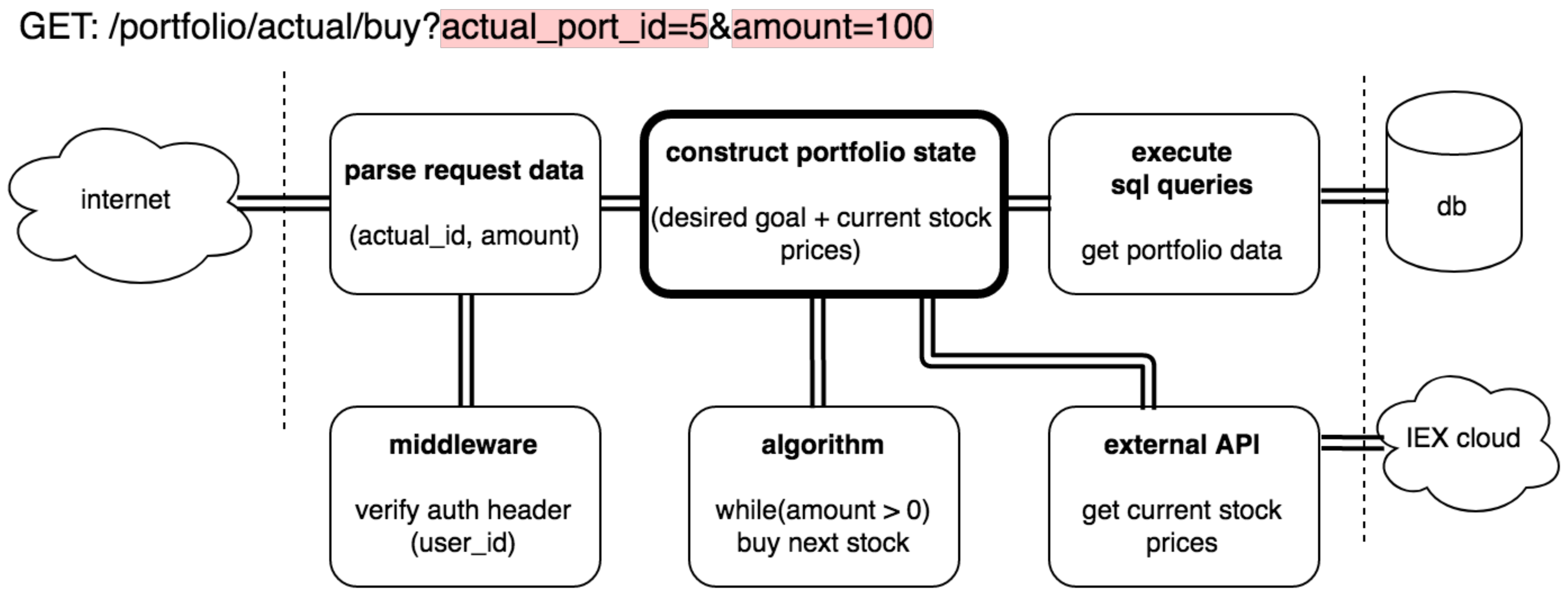
system boundaries

GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

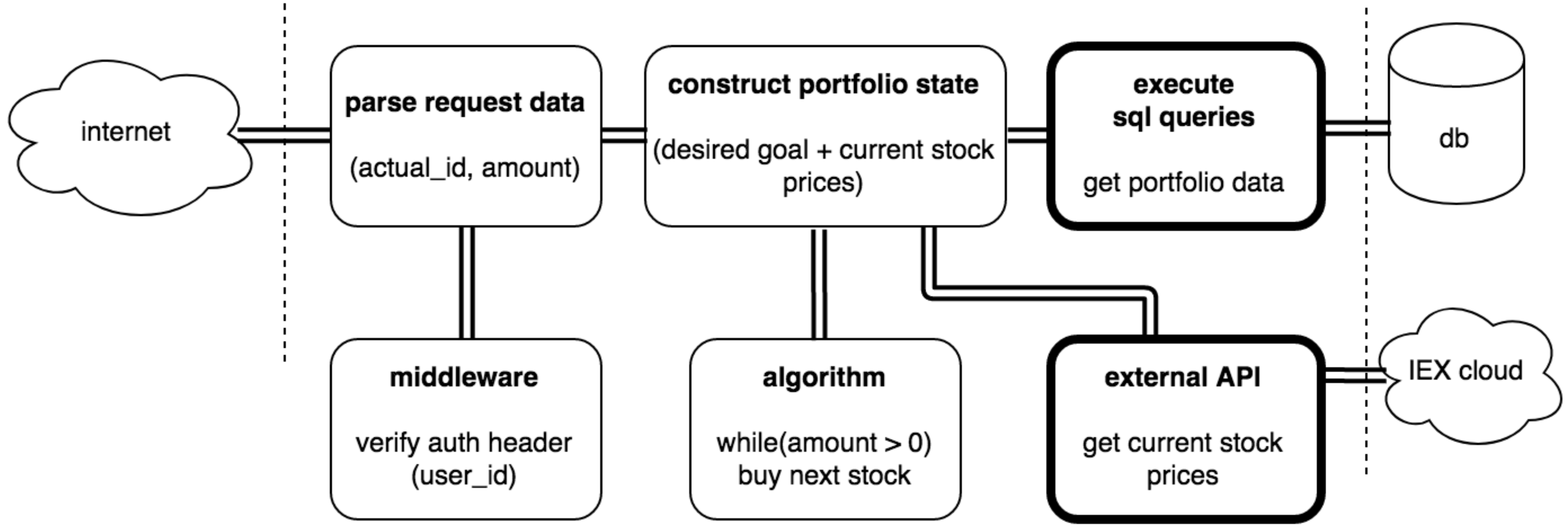
system boundaries


GET: /portfolio/actual/buy?actual_port_id=5&amount=100
internet
parse request data
(actual_id, amount)
construct portfolio state
(desired goal + current stock prices)
execute sql queries
get portfolio data
db
middleware
verify auth header (user_id)
algorithm
while(amount > 0) buy next stock
external API
get current stock prices
IEX cloud

designing with lifetimes(cont..)

system boundaries

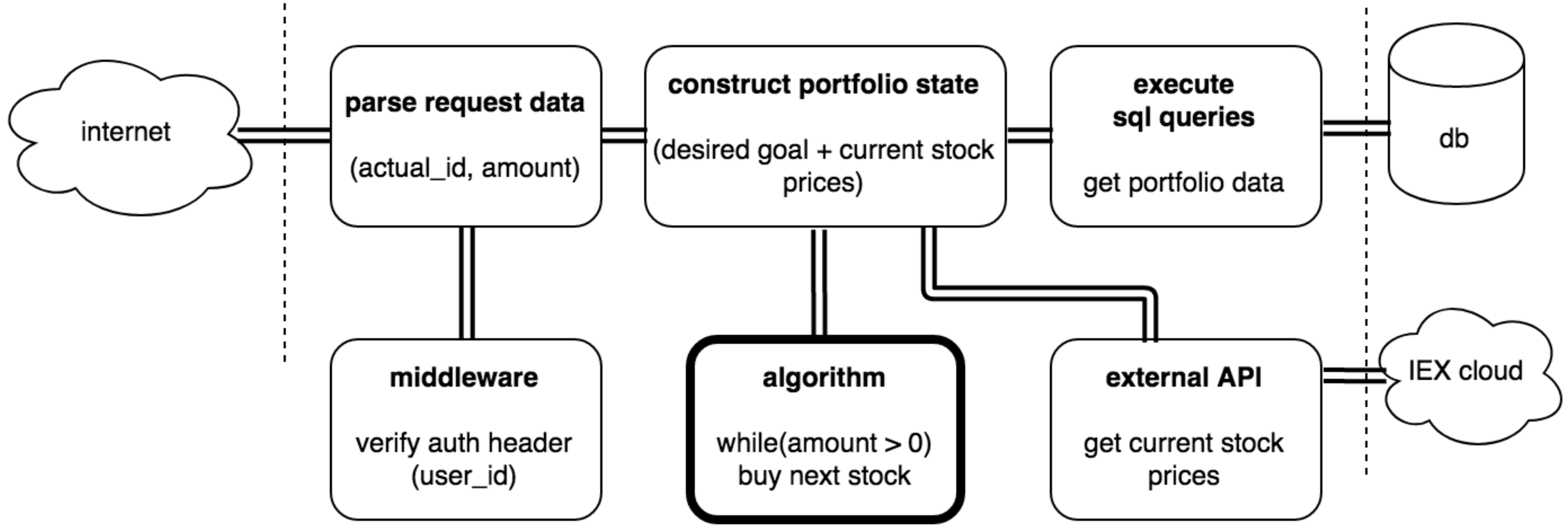
GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

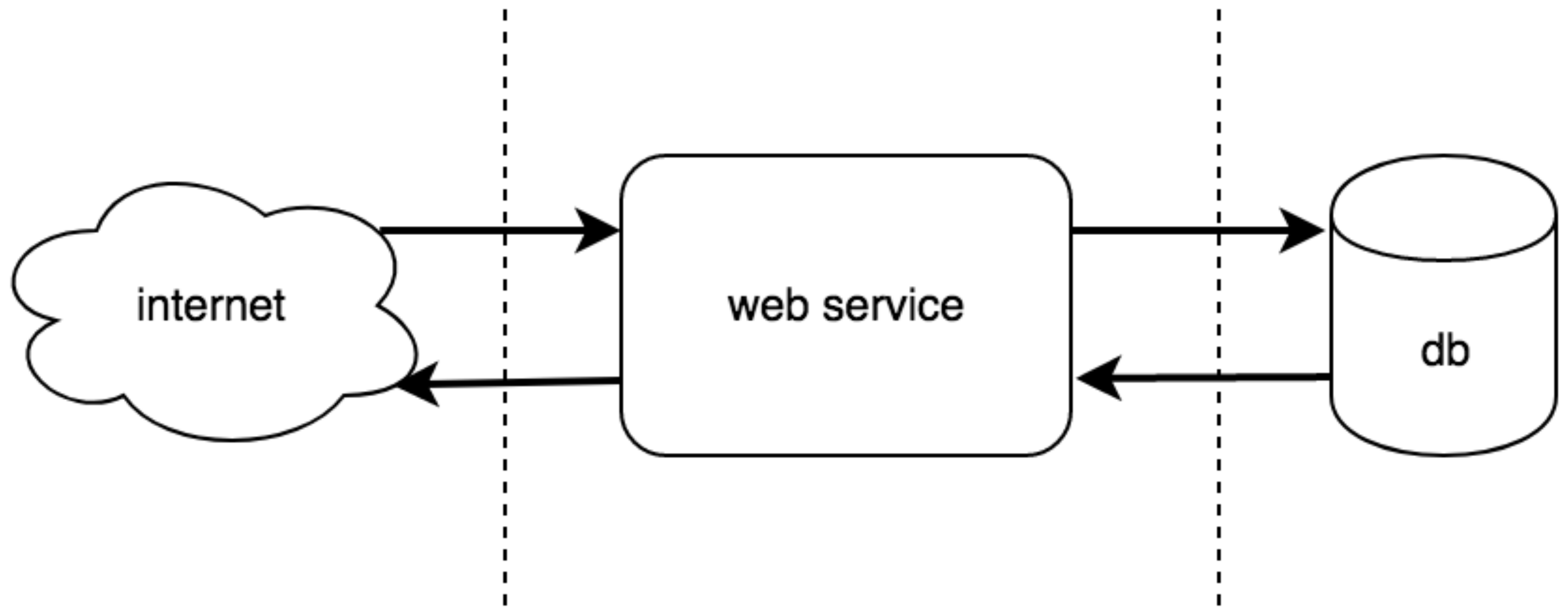
system boundaries

GET: /portfolio/actual/buy?actual_port_id=5&amount=100



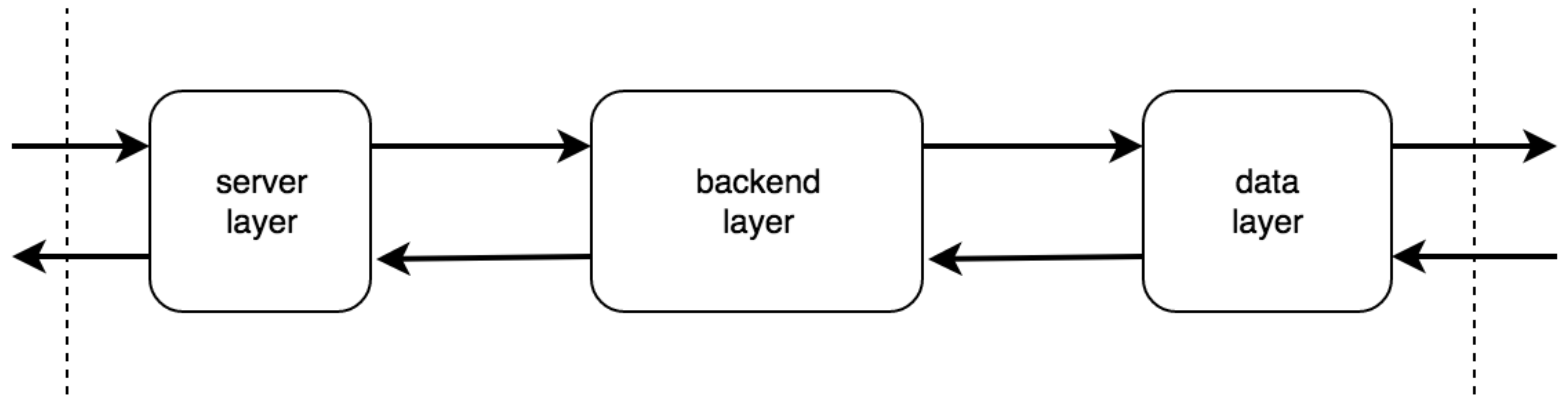
designing with lifetimes(cont..)

system boundaries(cont..)



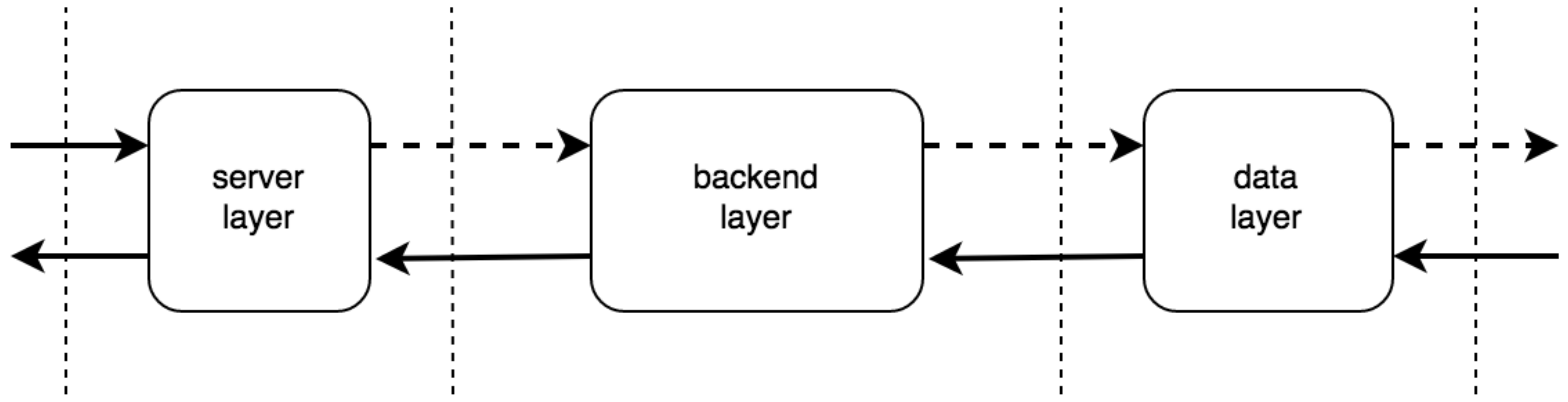
designing with lifetimes(cont..)

system boundaries(cont..)



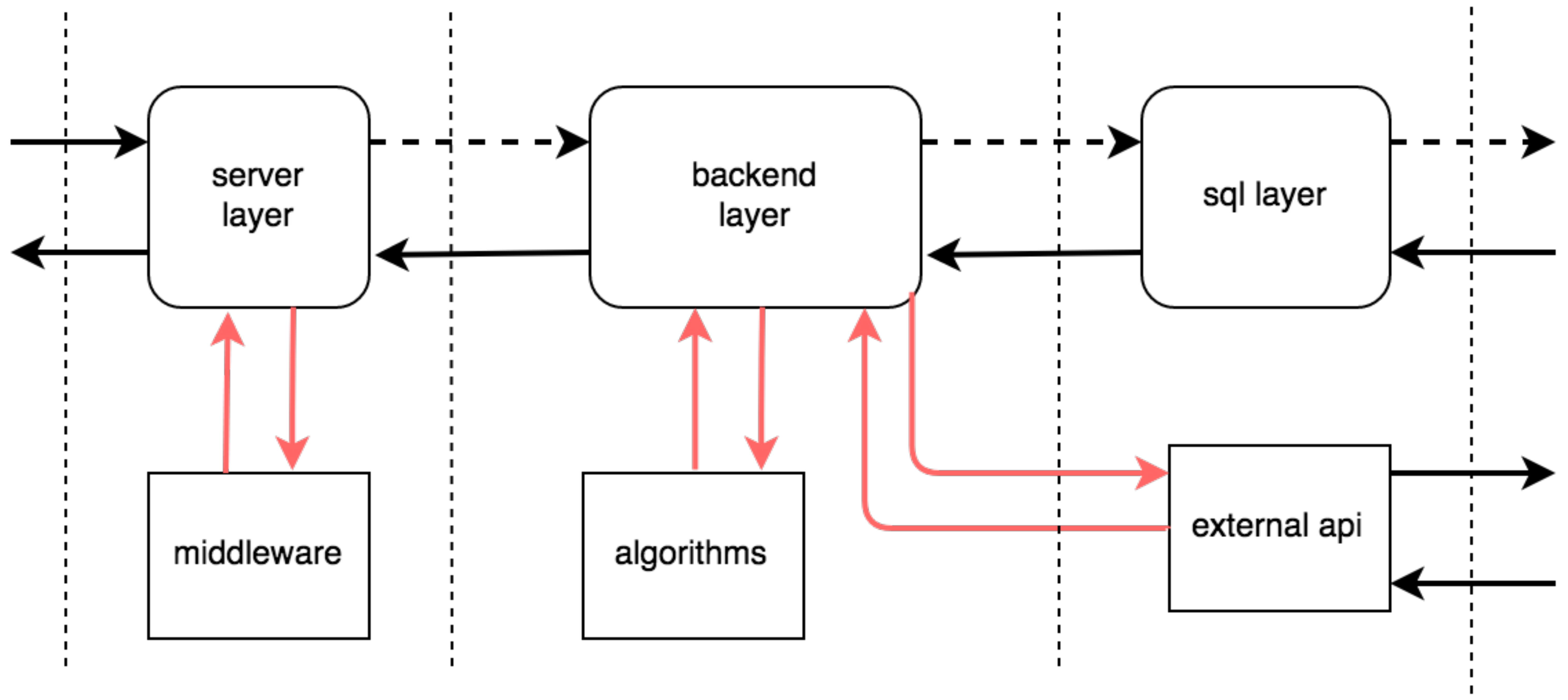
designing with lifetimes(cont..)

system boundaries(cont..)



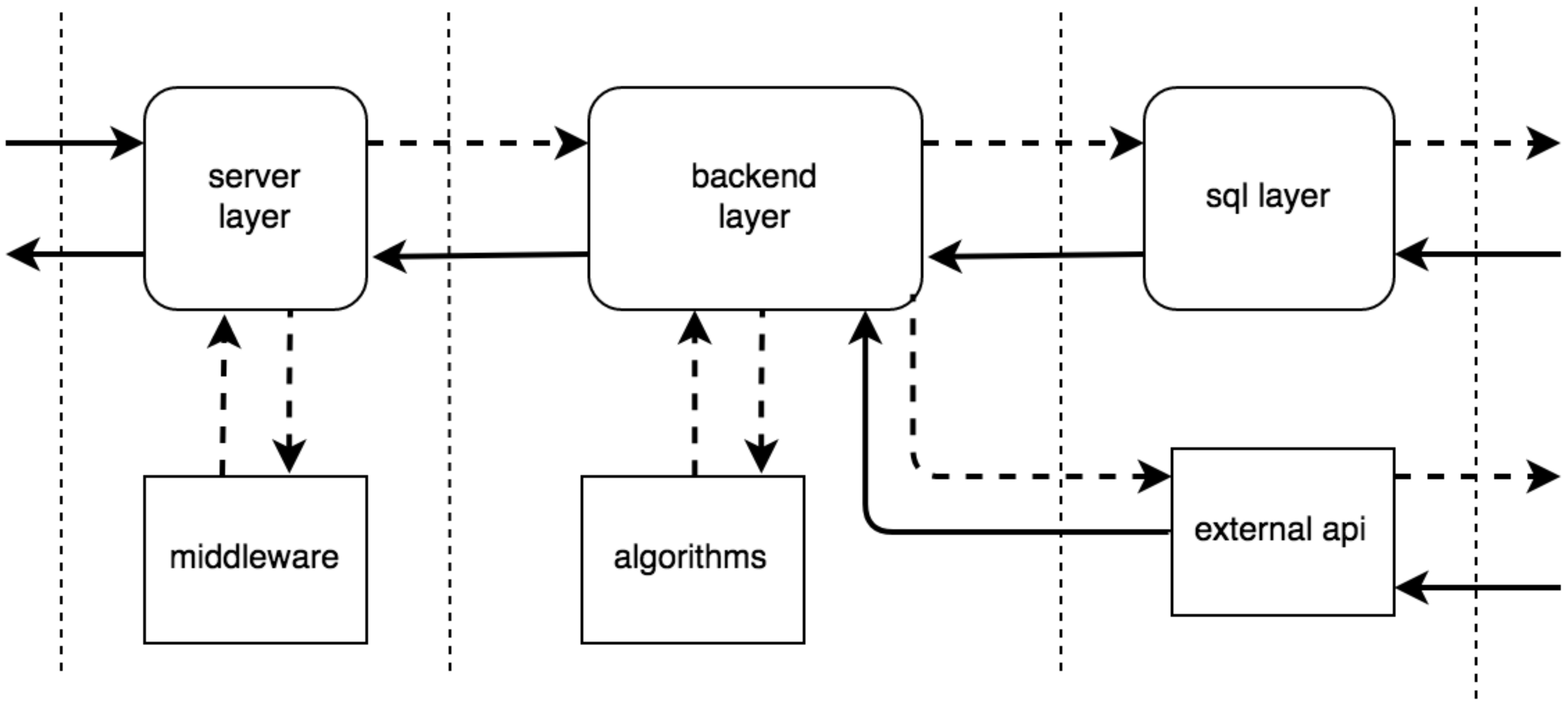
designing with lifetimes(cont..)

system boundaries(cont..)



designing with lifetimes(cont..)

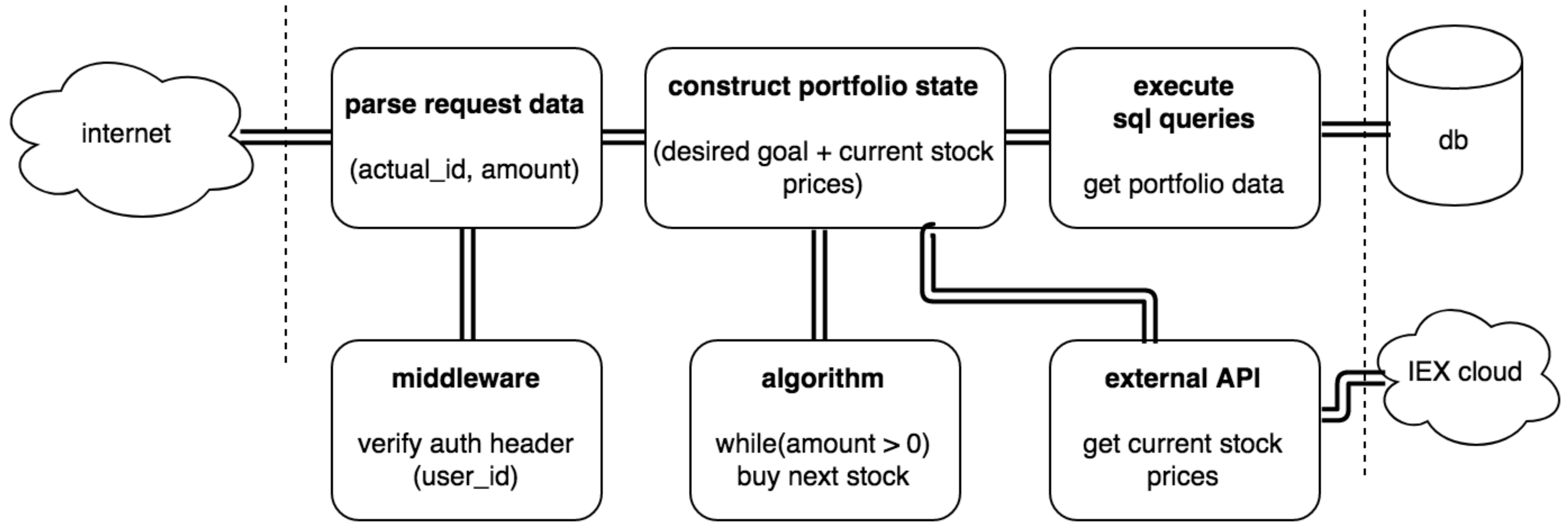
system boundaries(cont..)



designing with lifetimes(cont..)

system boundaries

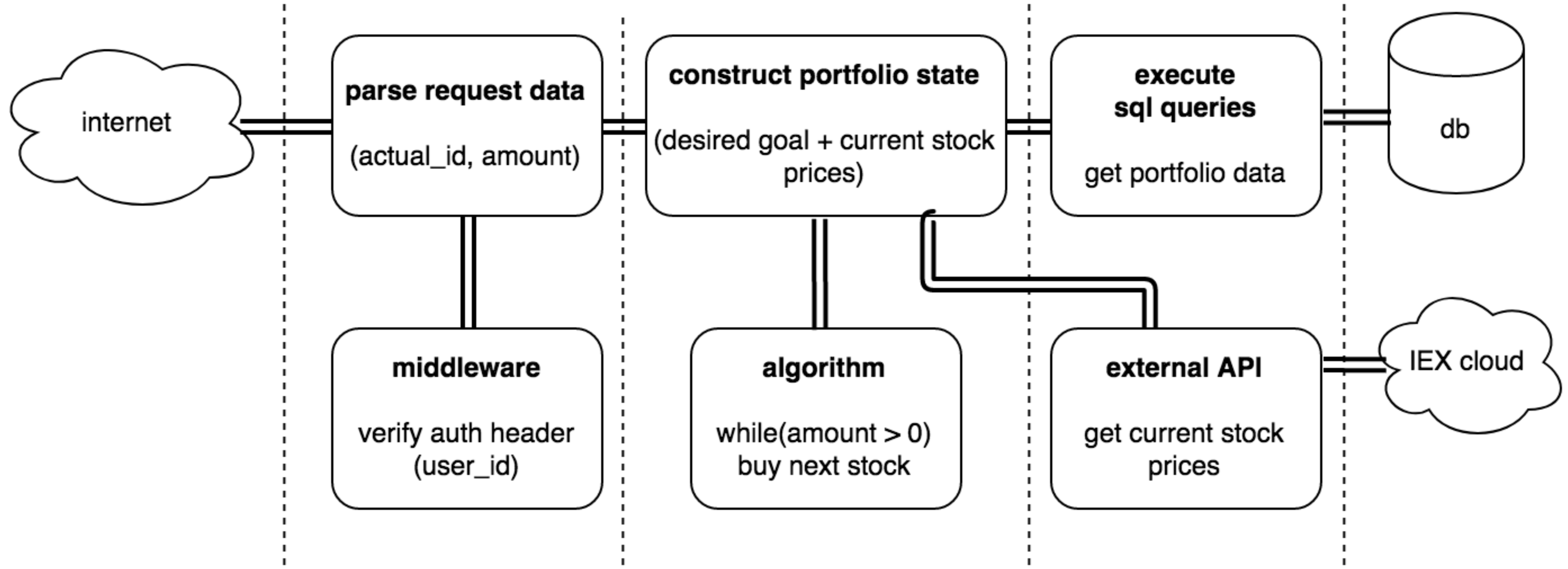
GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

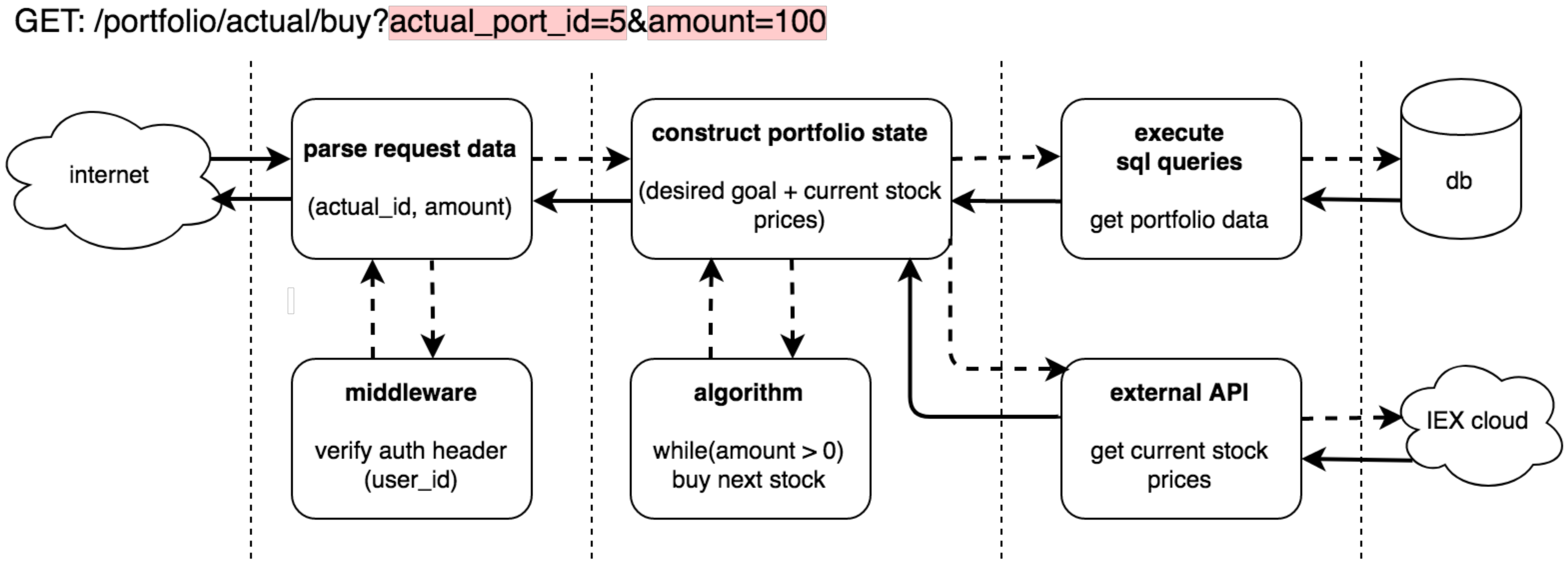
system boundaries

GET: /portfolio/actual/buy?actual_port_id=5&amount=100



designing with lifetimes(cont..)

system boundaries

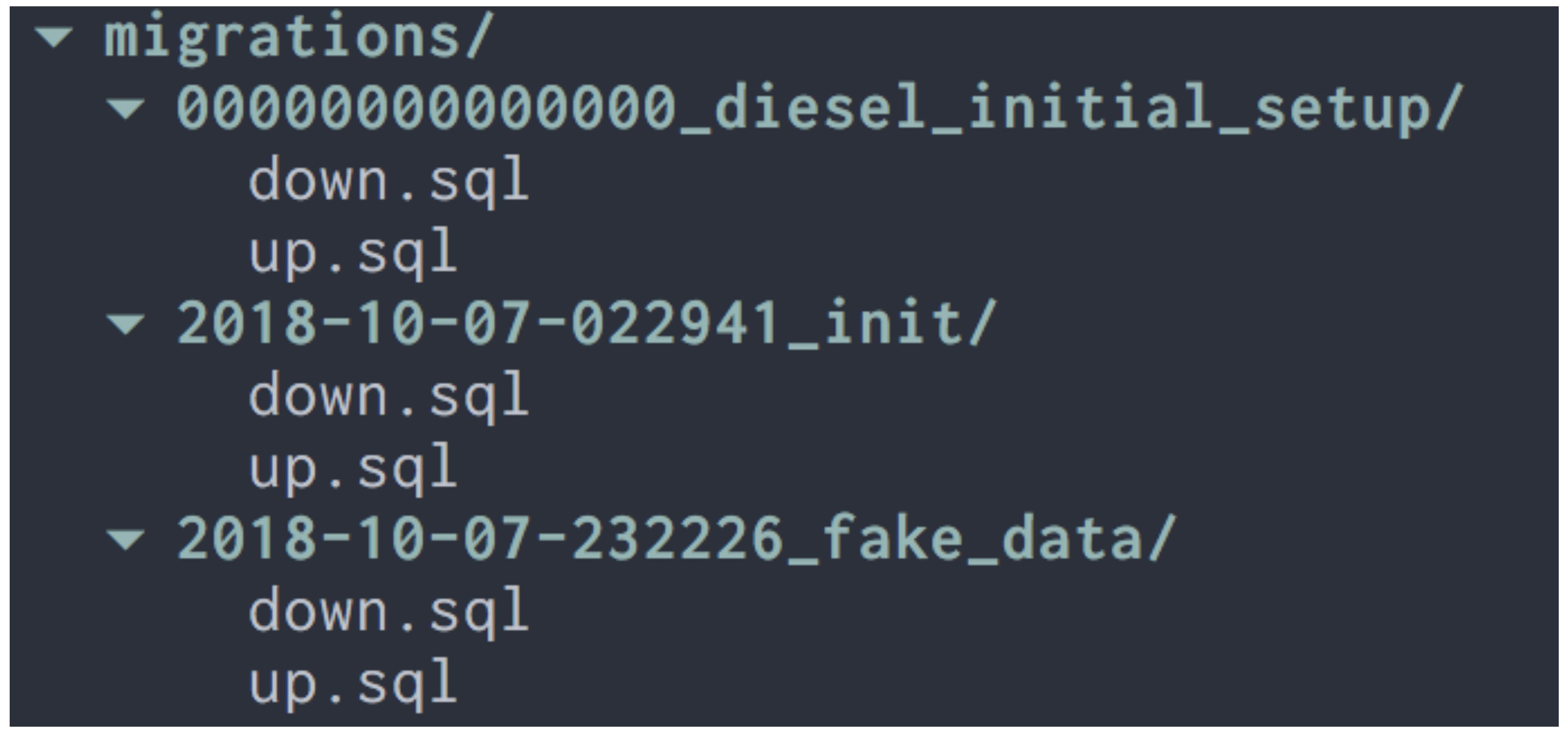


db management

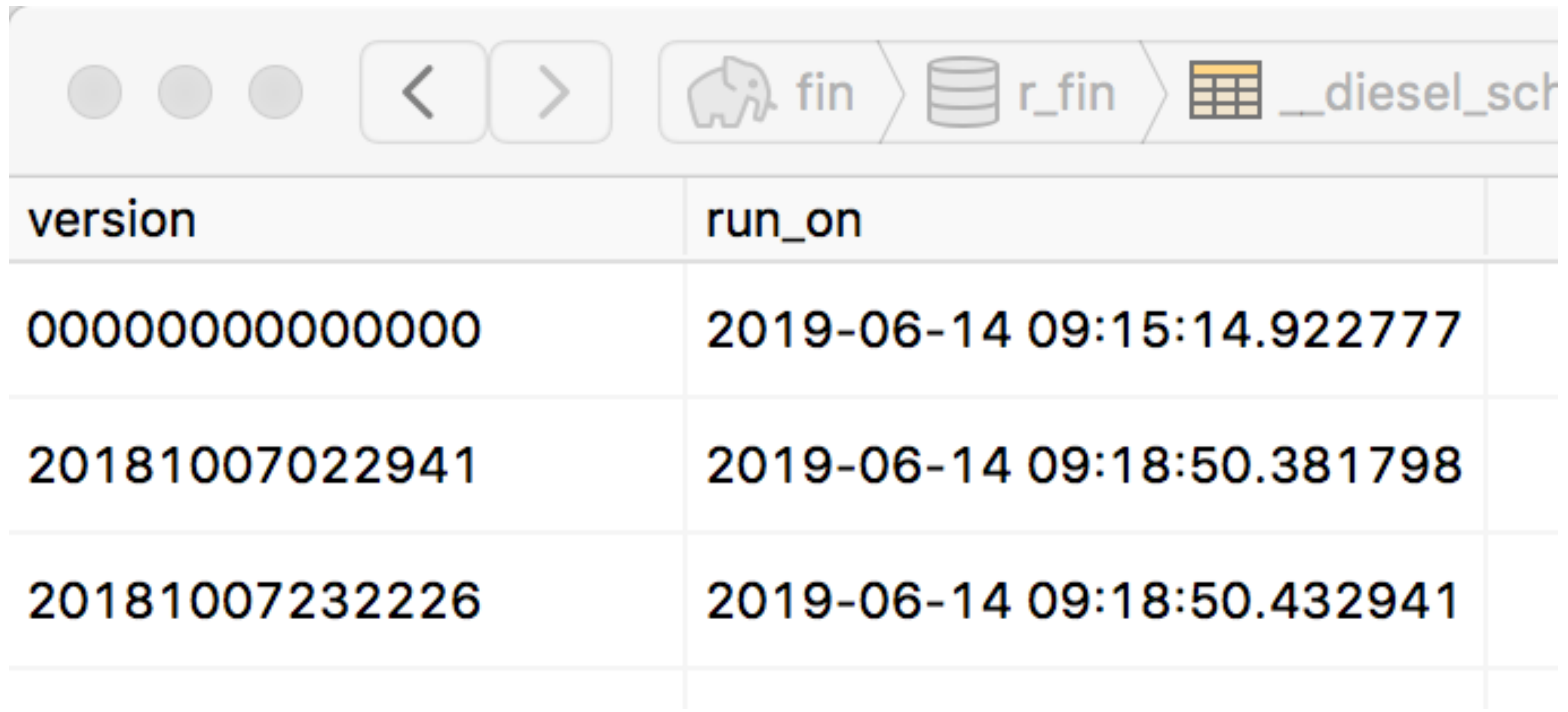
db management (cont..)

migrations with `diesel_cli`

- *manage database schema*
    - *evolve schema*
    - *code review and test schema changes*



```
▾ 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
```



version	run_on
00000000000000	2019-06-14 09:15:14.922777
20181007022941	2019-06-14 09:18:50.381798
20181007232226	2019-06-14 09:18:50.432941

db management (cont..)

ORM vs raw SQL

ORM (Diesel)	raw SQL
• pro: queries typed checked at compile time!	• pro: simply write the SQL you want
• con: can be difficult to write complex queries and need to learn a new framework	• con: need to write queries manually which can be error prone

db management (cont..)

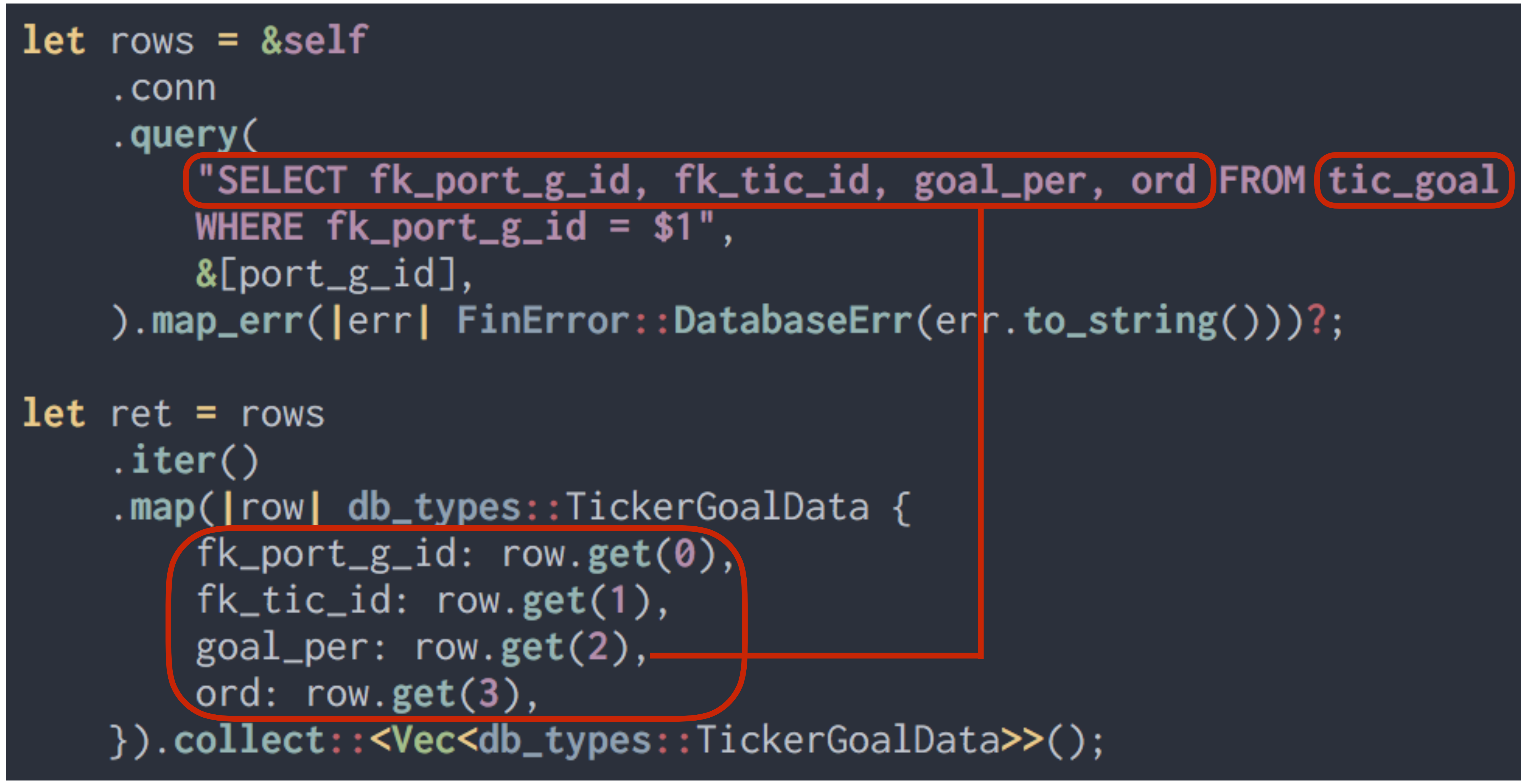
postgres crate

- flexibility of raw SQL

  but

- error prone

- difficult to maintain

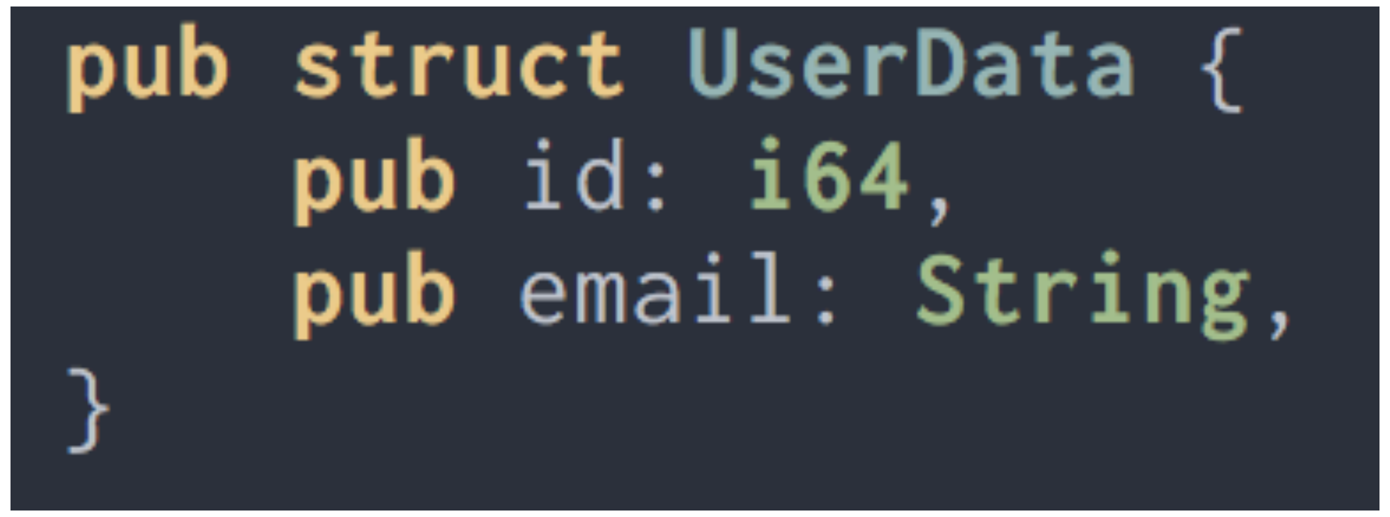


```
let rows = &self
    .conn
    .query(
        "SELECT fk_port_g_id, fk_tic_id, goal_per, ord FROM tic_goal
        WHERE fk_port_g_id = $1",
        &[port_g_id],
    ).map_err(|err| FinError::DatabaseErr(err.to_string()))?;

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>>();
```

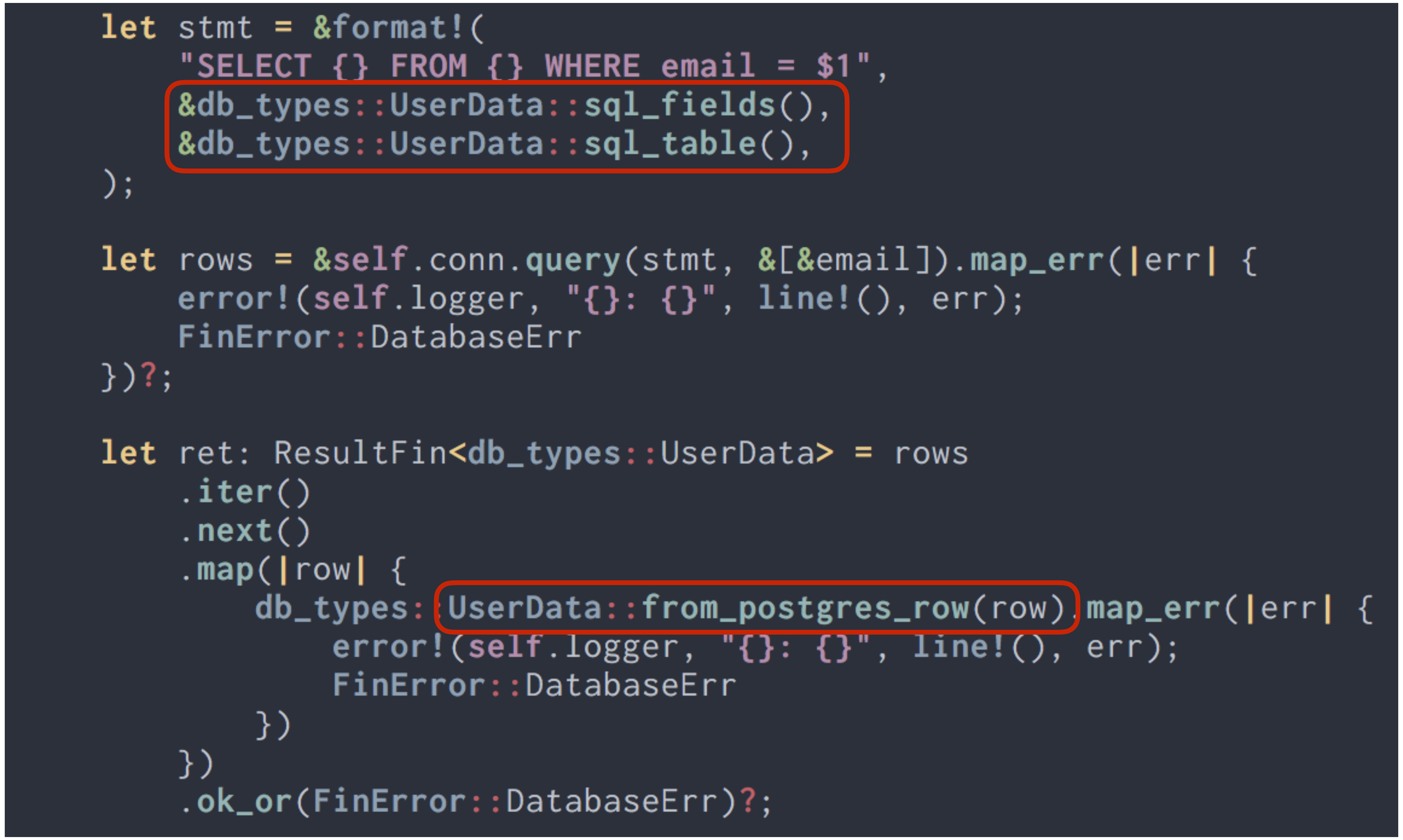
db management (cont..)

postgres crate

- ideal SQL



```rust
pub struct UserData {
    pub id: i64,
    pub email: String,
}
```

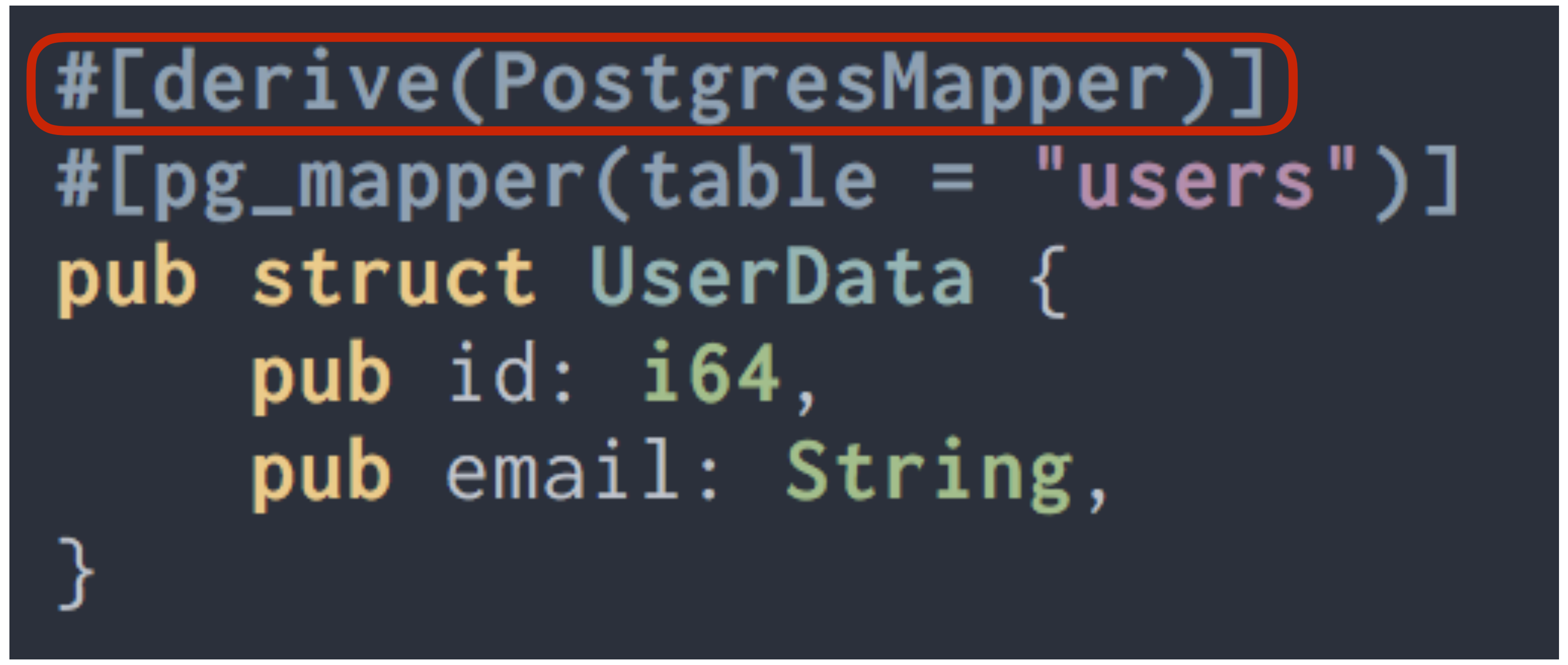


```rust
let stmt = &format!(
    "SELECT {} FROM {} WHERE email = $1",
    &db_types::UserData::sql_fields(),
    &db_types::UserData::sql_table(),
);

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)?;
```

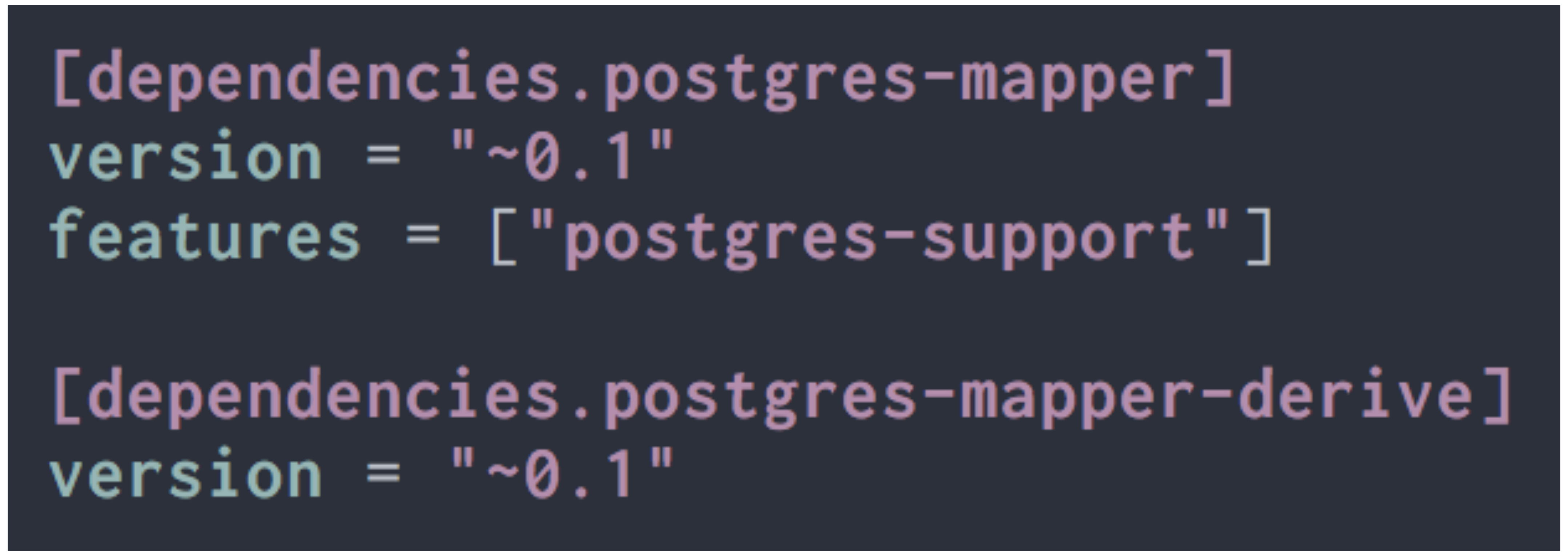
db management (cont..)

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,
}
```

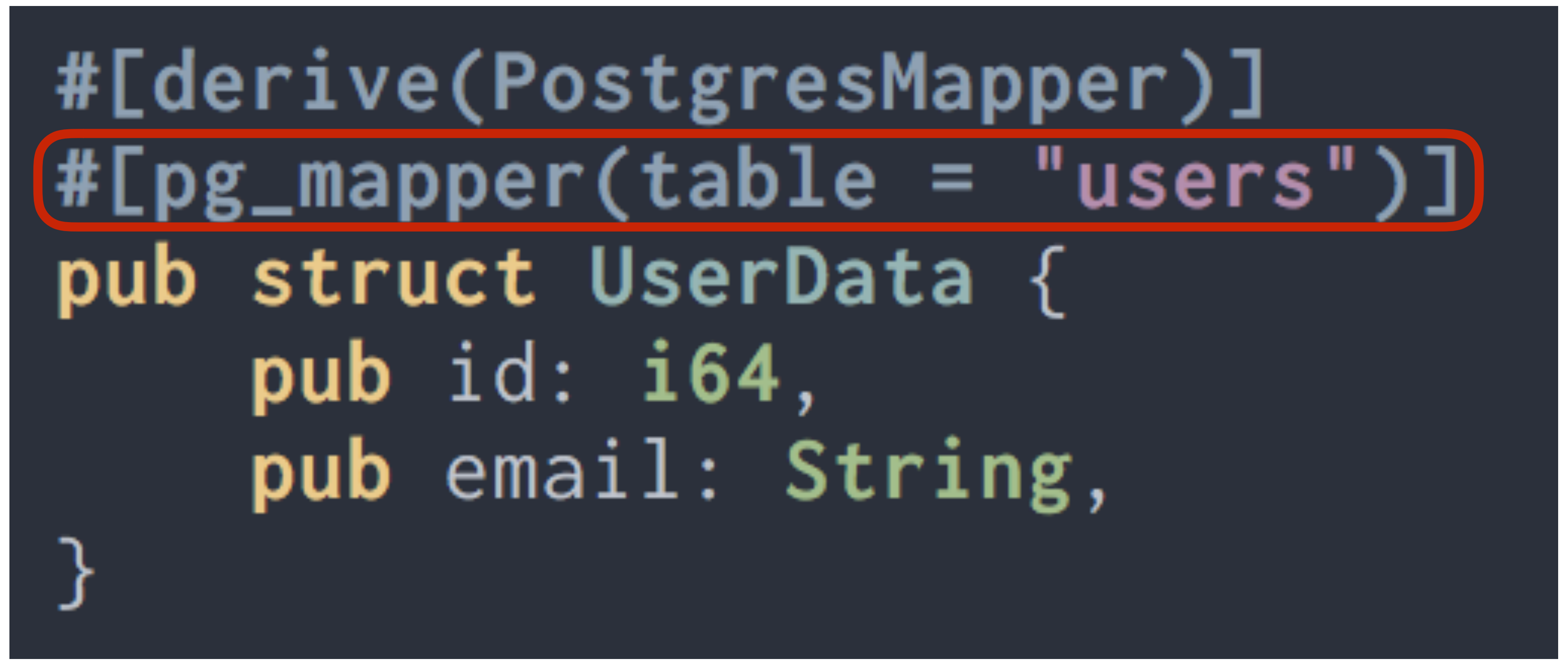


```toml
[dependencies.postgres-mapper]
version = "~0.1"
features = ["postgres-support"]

[dependencies.postgres-mapper-derive]
version = "~0.1"
```

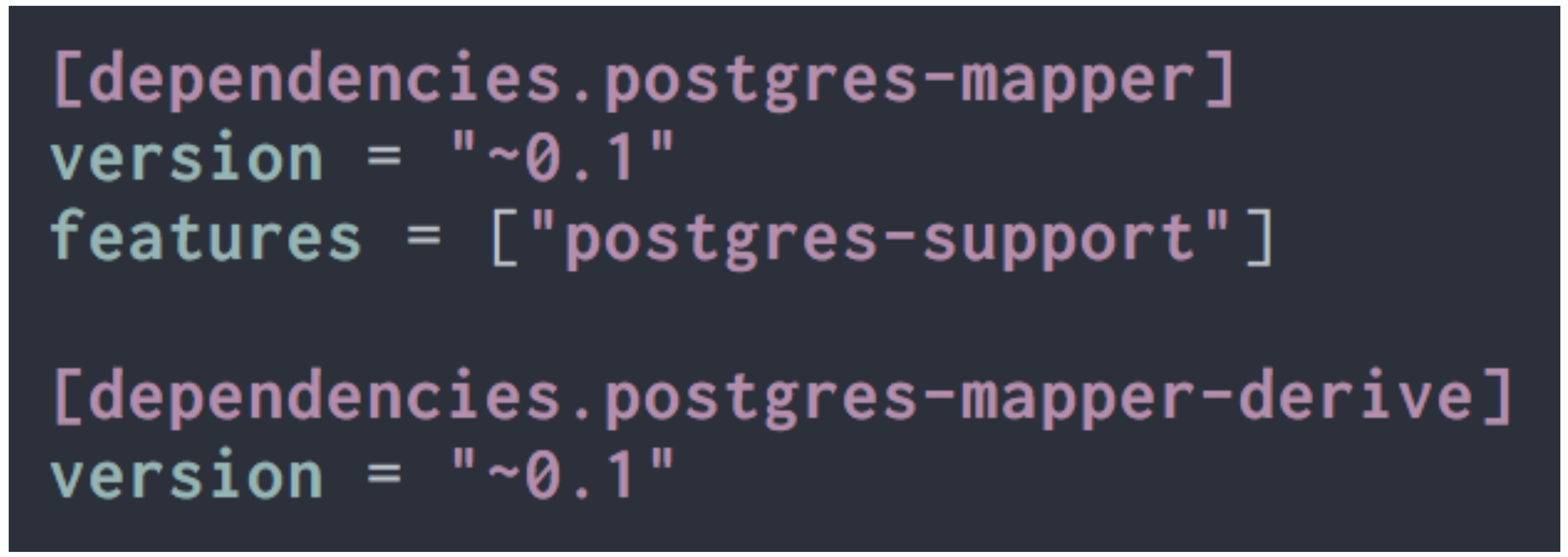
db management (cont..)

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**”



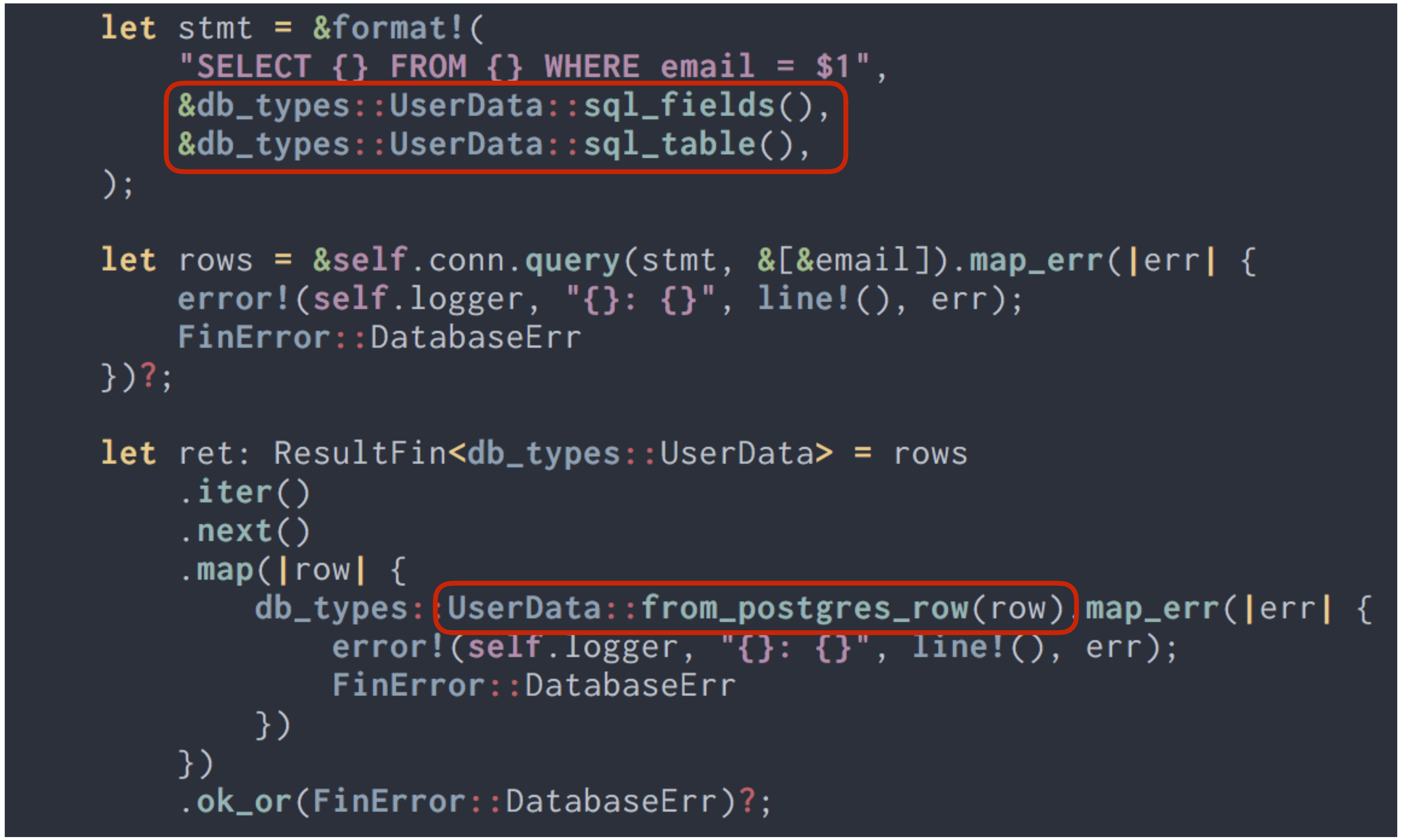
```rust
#[derive(PostgresMapper)]
#[pg_mapper(table = "users")]
pub struct UserData {
    pub id: i64,
    pub email: String,
}
```



```toml
[dependencies.postgres-mapper]
version = "~0.1"
features = ["postgres-support"]

[dependencies.postgres-mapper-derive]
version = "~0.1"
```

db management (cont..)

postgres crate

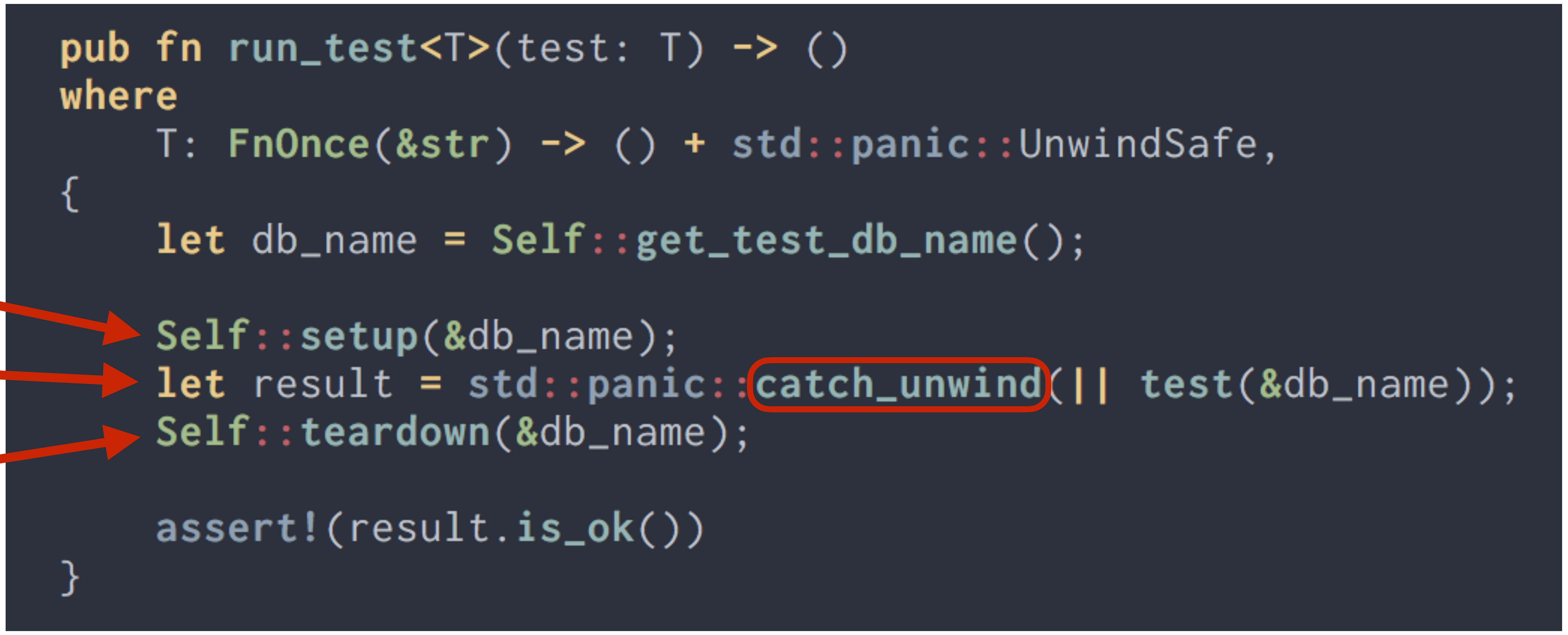


```rust
let stmt = &format!(
    "SELECT {} FROM {} WHERE email = $1",
    &db_types::UserData::sql_fields(),
    &db_types::UserData::sql_table(),
);

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)?;
```

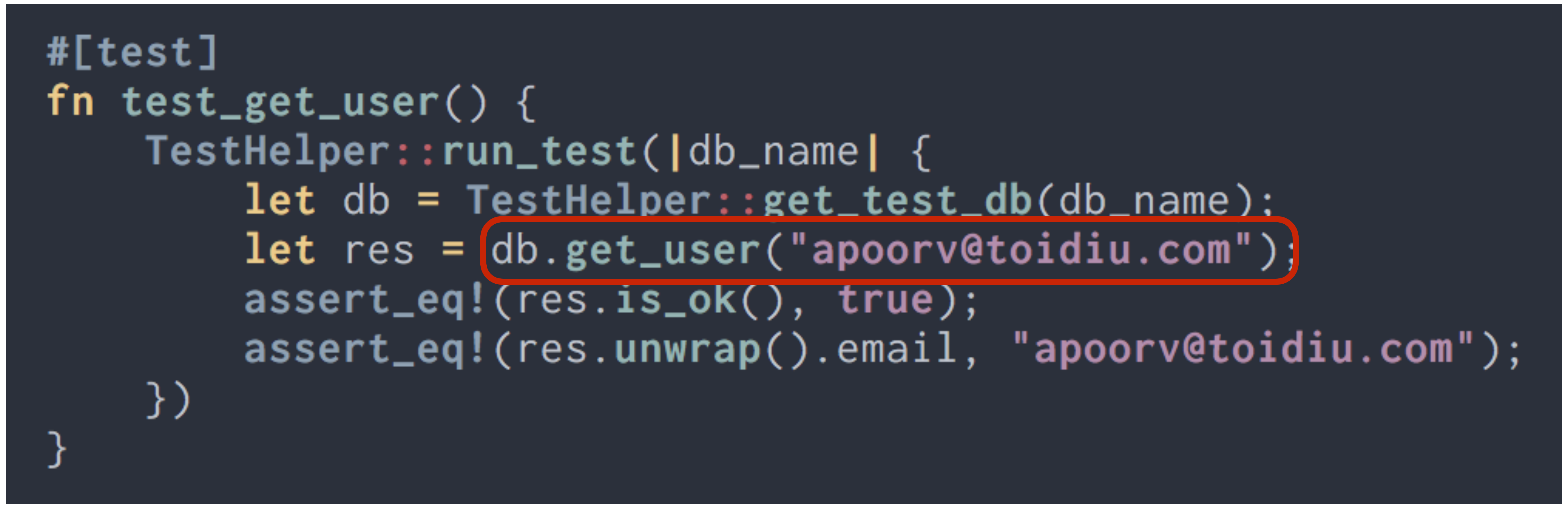
db management(cont..)

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())
}
```

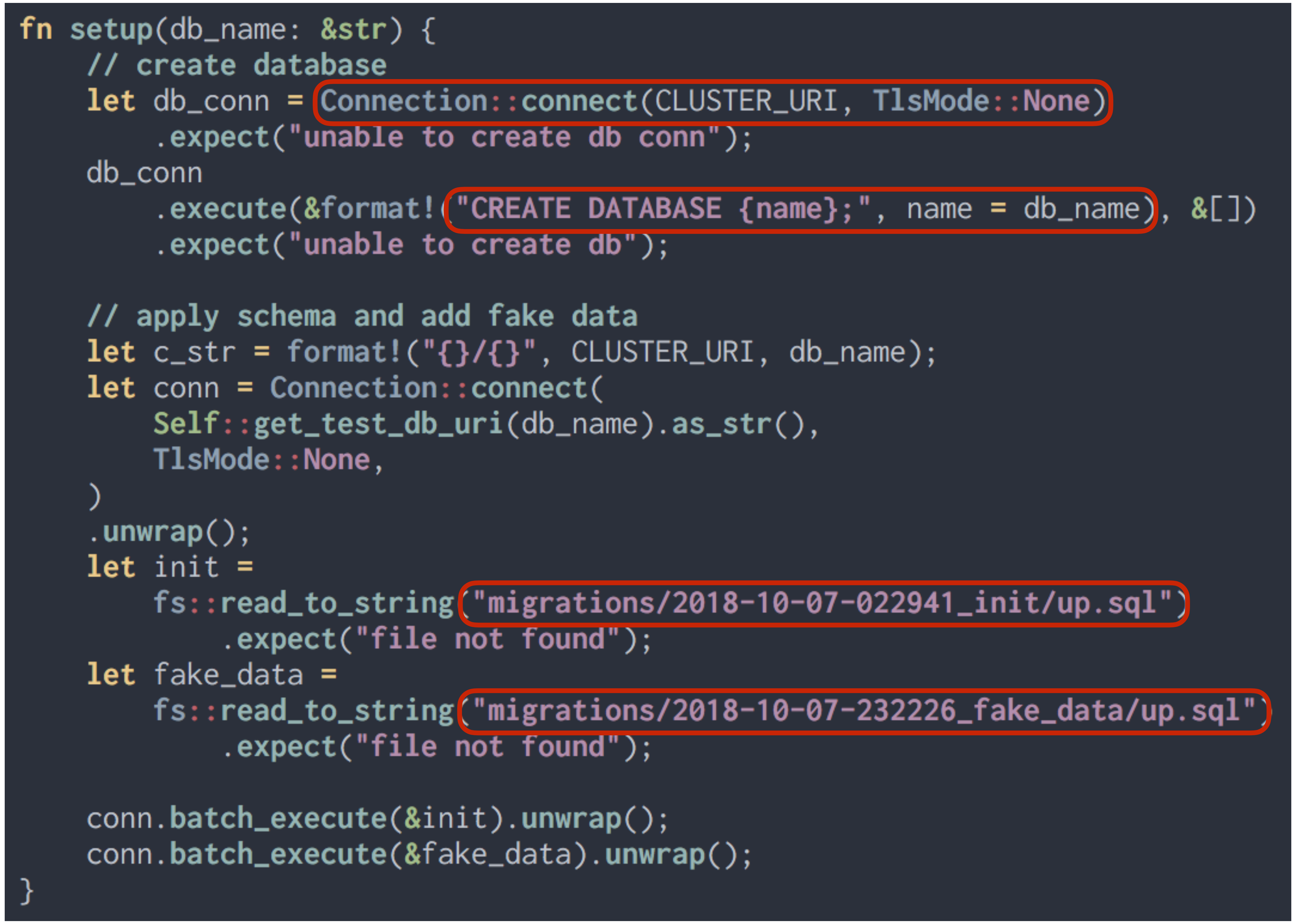


```rust
#[test]
fn test_get_user() {
    TestHelper::run_test(|db_name| {
        let db = TestHelper::get_test_db(db_name);
        let res = db.get_user("apoorv@toidiu.com");
        assert_eq!(res.is_ok(), true);
        assert_eq!(res.unwrap().email, "apoorv@toidiu.com");
    })
}
```

db management(cont..)

db testing

- setup DB for testing
  - get a connection (needs a 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();
}
```

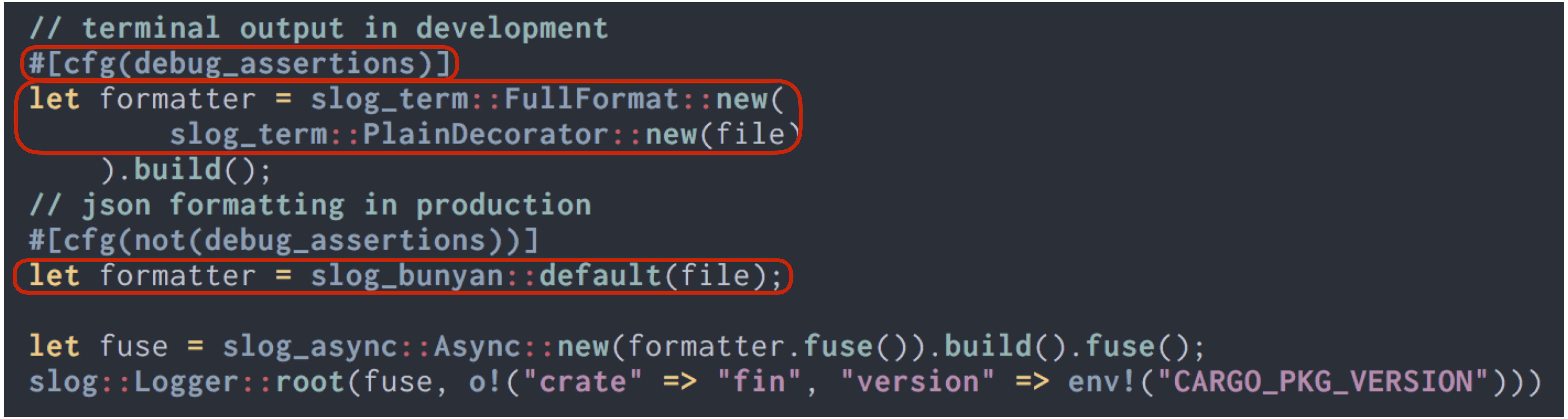
logging

logging(cont..)

slog composable

- composable plugin model `**trait Drain**`

json vs plain async vs sync file vs network



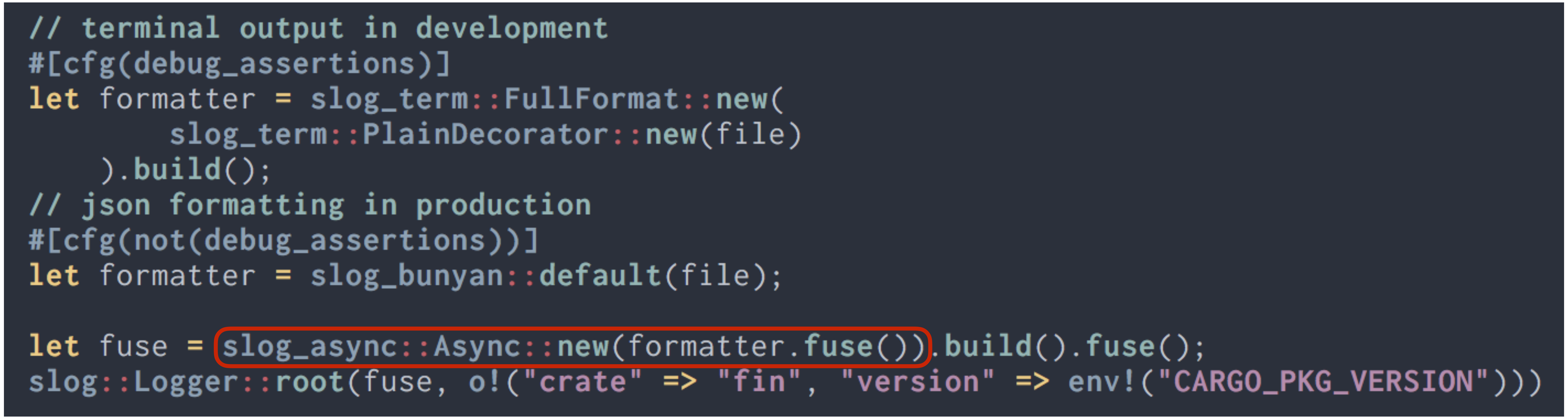
```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

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 composable

- composable plugin model `**trait Drain**`

json vs plain **async vs sync** file vs network



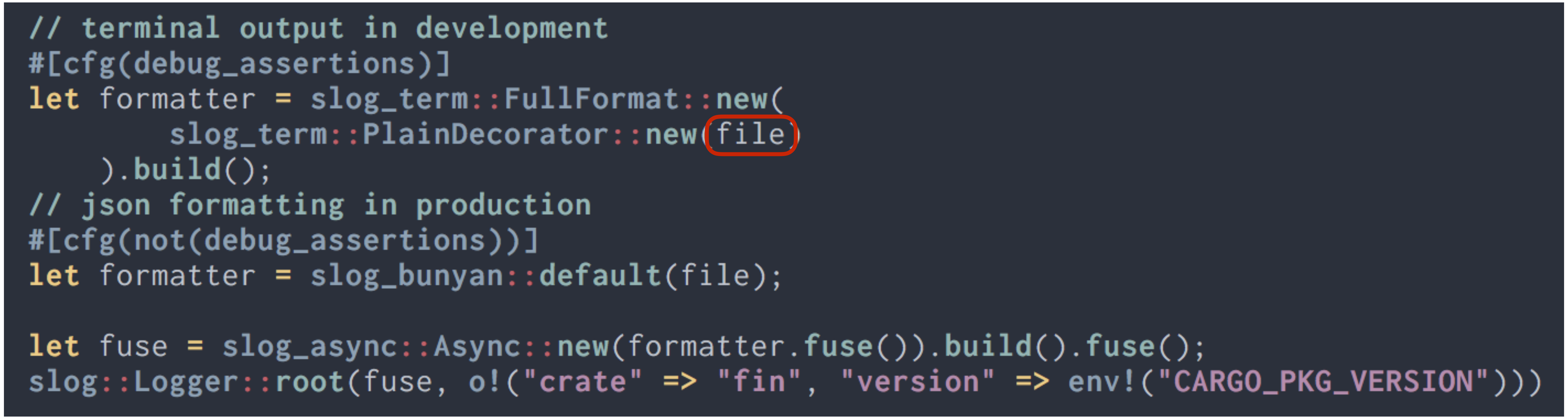
```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

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 composable

- composable plugin model `**trait Drain**`

json vs plain   async vs sync   **file vs network**

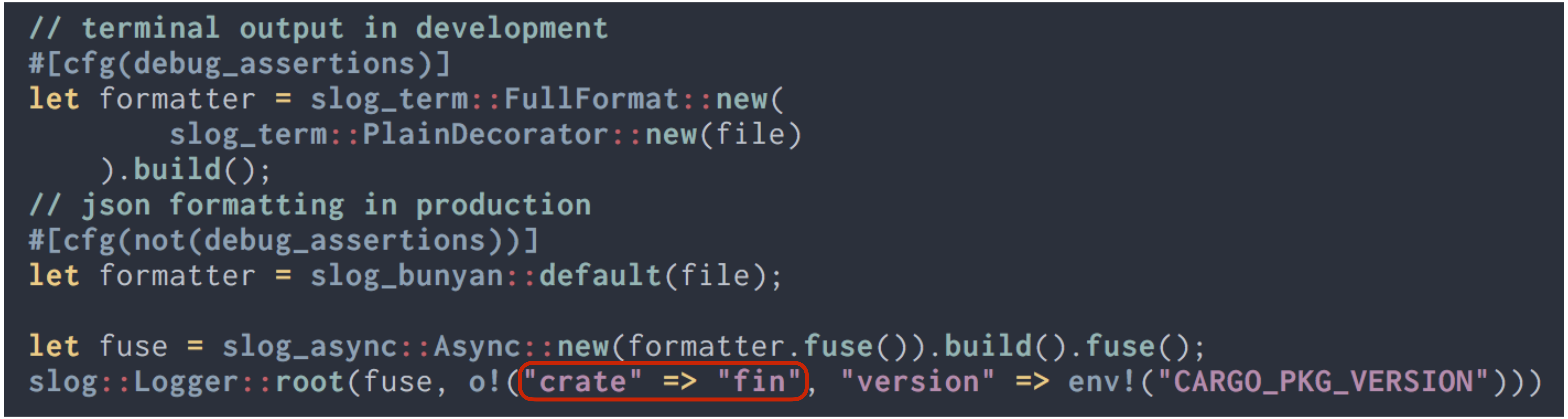


```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

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

- log data should be **machine searchable vs writing complex regex**
  - think key-value pairs
  - ex: filter logs by 'error codes', 'app version', 'req id'

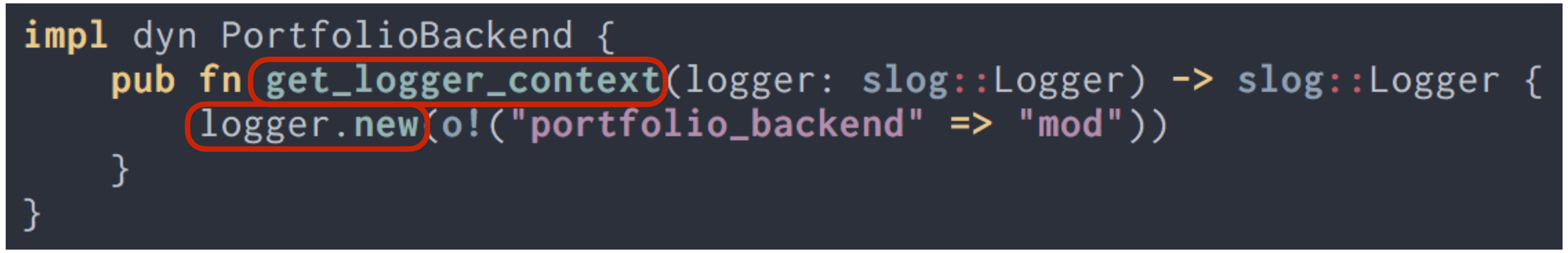


```
// terminal output in development
#[cfg(debug_assertions)]
let formatter = slog_term::FullFormat::new(
        slog_term::PlainDecorator::new(file)
    ).build();
// json formatting in production
#[cfg(not(debug_assertions))]
let formatter = slog_bunyan::default(file);

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

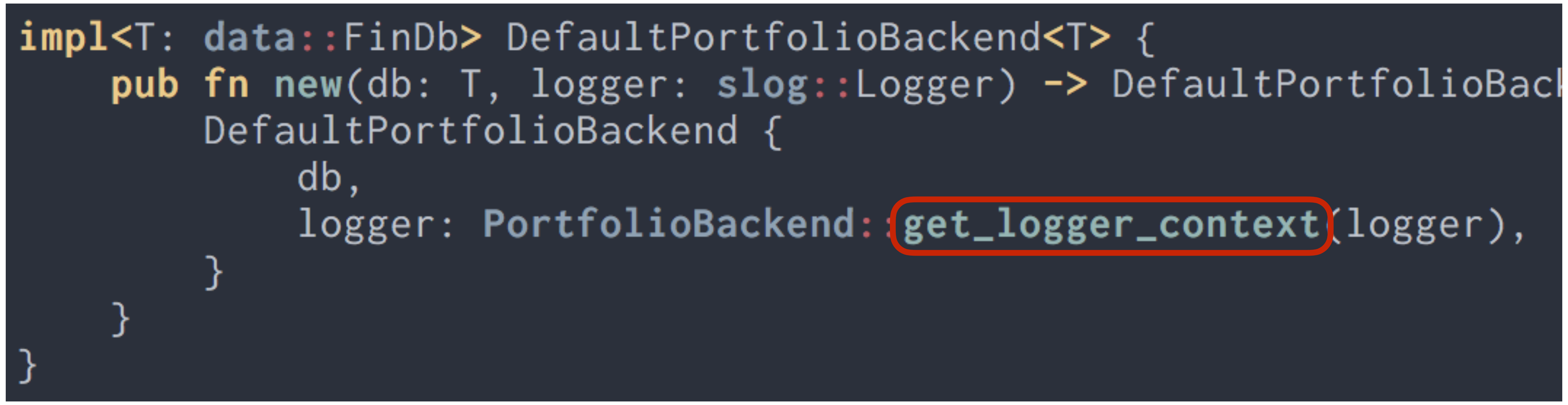
logging(cont..)

slog contextual

- give context around error
- trace code path
- Logger is cheap to clone



```
impl dyn PortfolioBackend {
    pub fn get_logger_context(logger: slog::Logger) -> slog::Logger {
        logger.new(o!("portfolio_backend" => "mod"))
    }
}
```

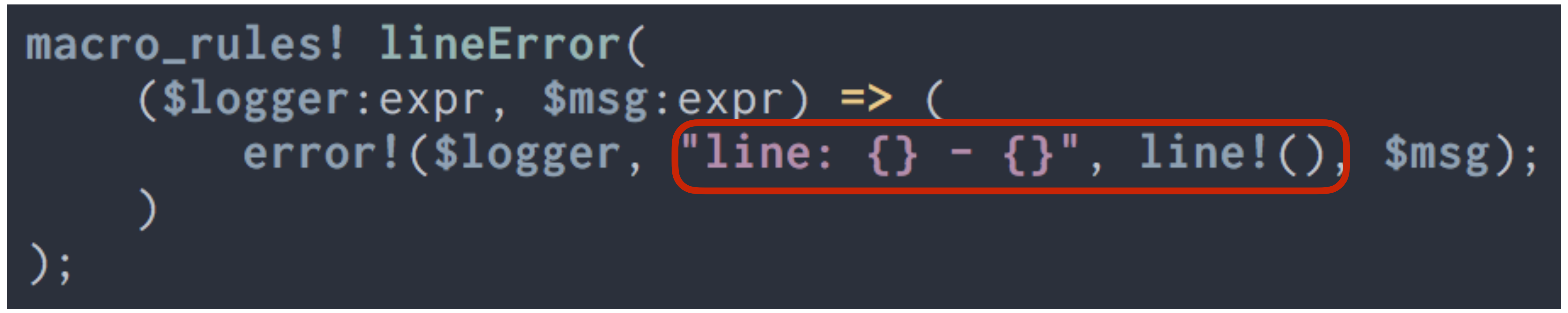


```
impl<T: data::FinDb> DefaultPortfolioBackend<T> {
    pub fn new(db: T, logger: slog::Logger) -> DefaultPortfolioBac
        DefaultPortfolioBackend {
            db,
            logger: PortfolioBackend::get_logger_context(logger),
        }
    }
}
```

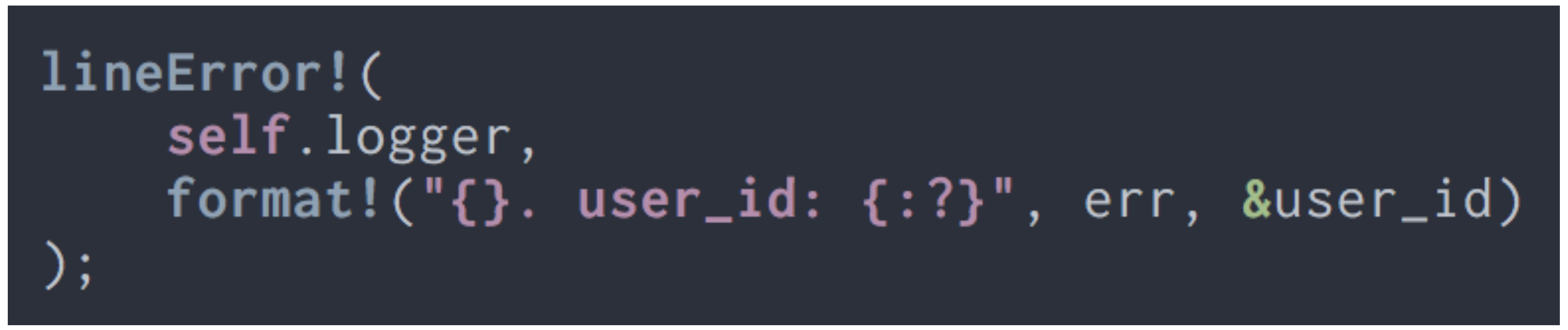
logging(cont..)

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);
    )
);
```



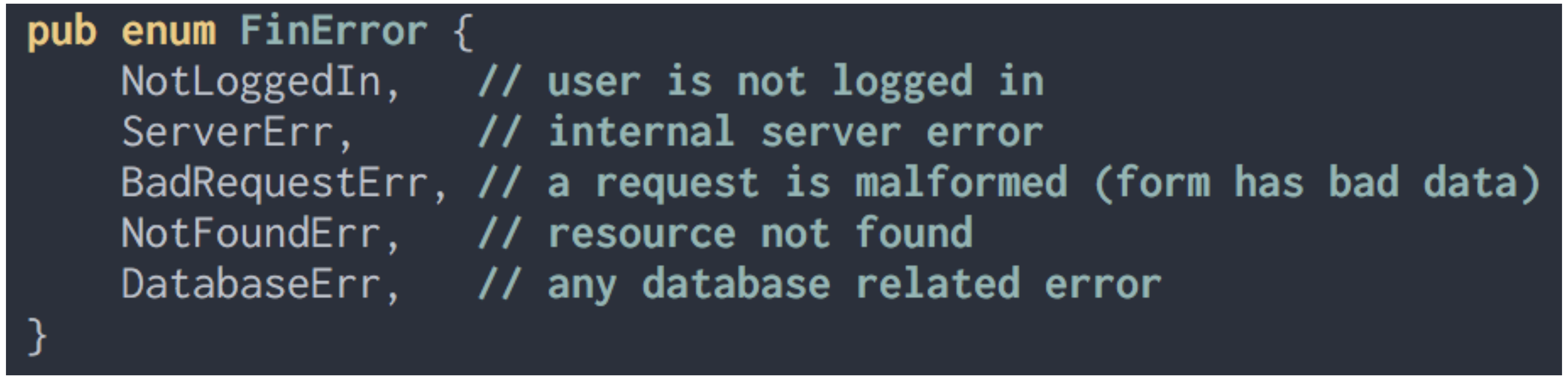
```
lineError!(
    self.logger,
    format!("{}. user_id: {:?}", err, &user_id)
);
```

code hardening

code hardening(cont..)

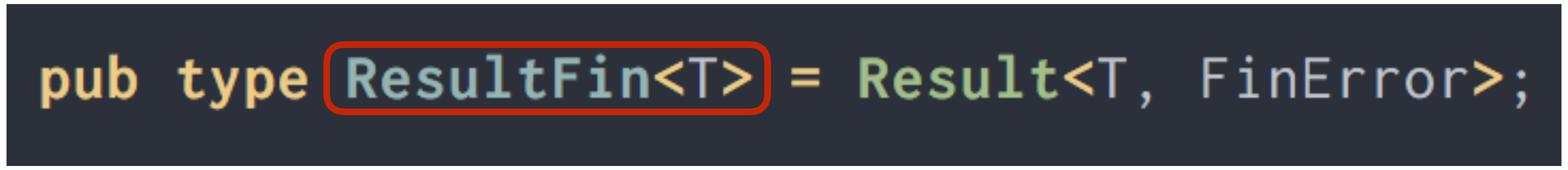
error handling

- declare global **AppError**(FinError) enum



```rust
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
}
```

- declare type alias **AppResult**(ResultFin)



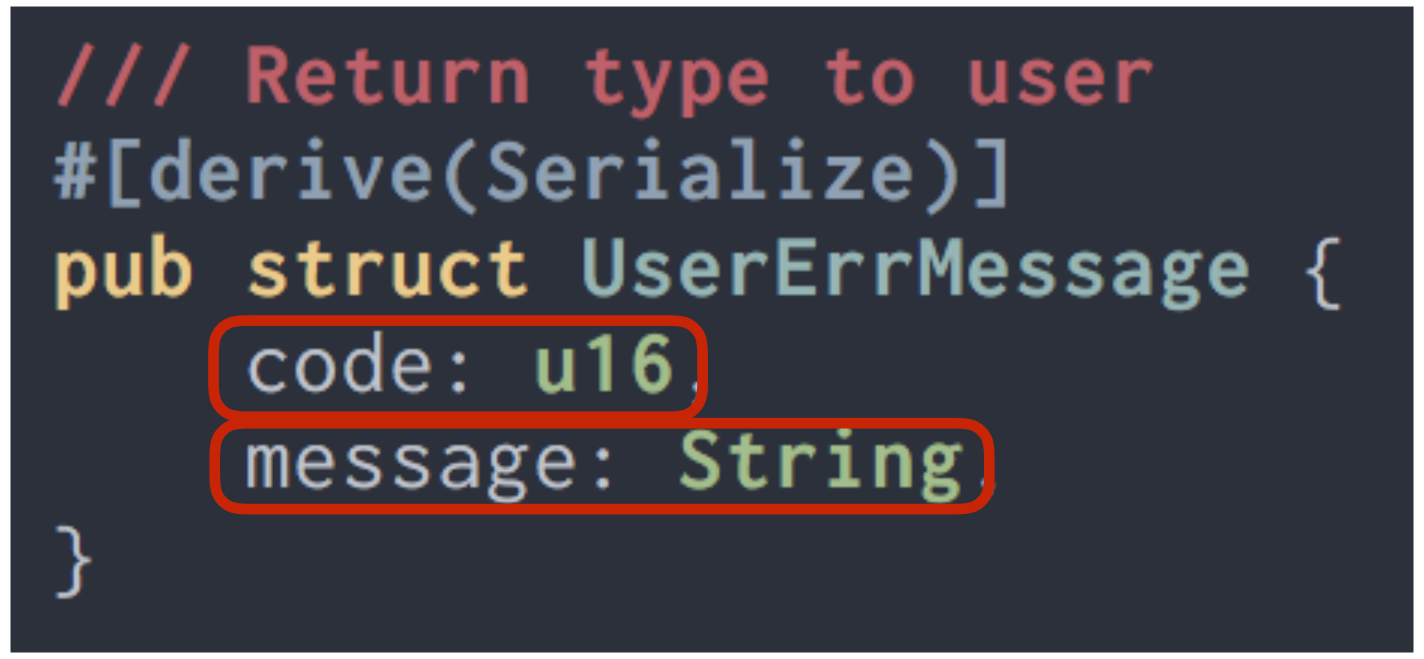
```rust
pub type ResultFin<T> = Result<T, FinError>;
```

- **all functions that return Result should only return AppResult!!**

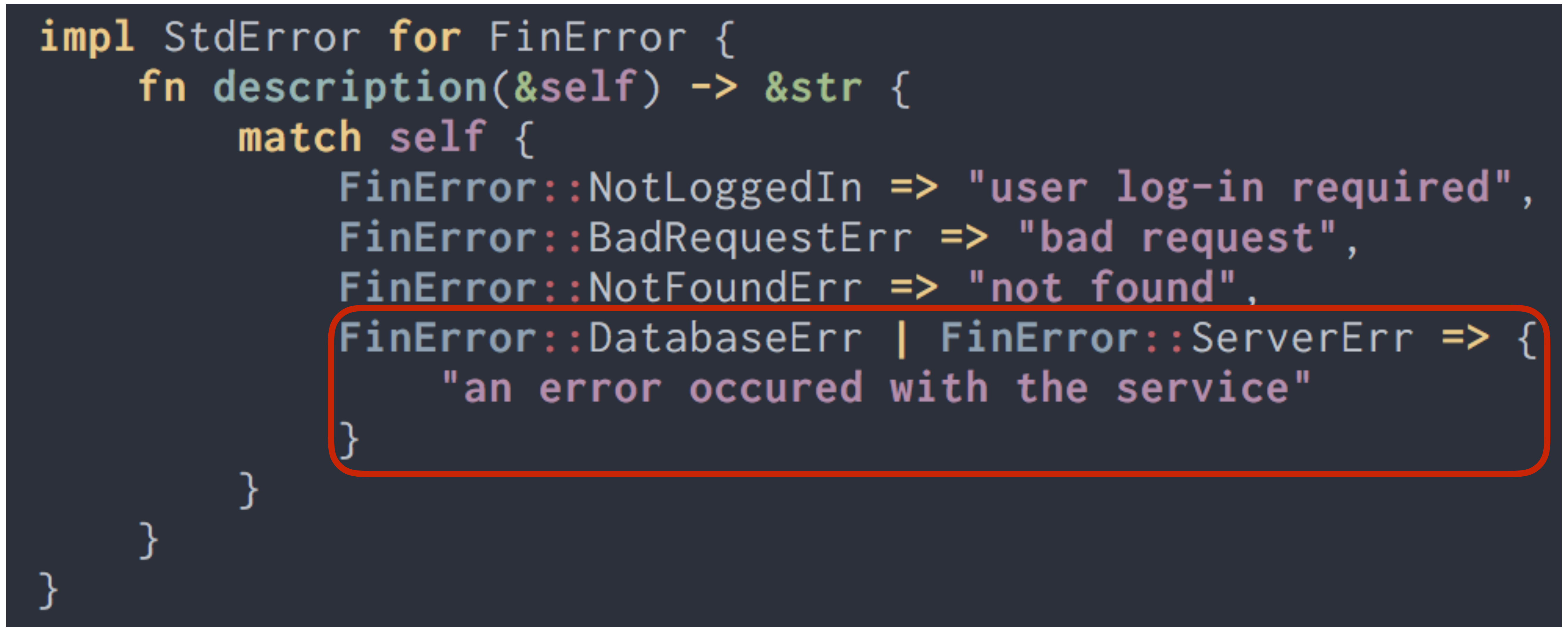
code hardening(cont..)

user error msg

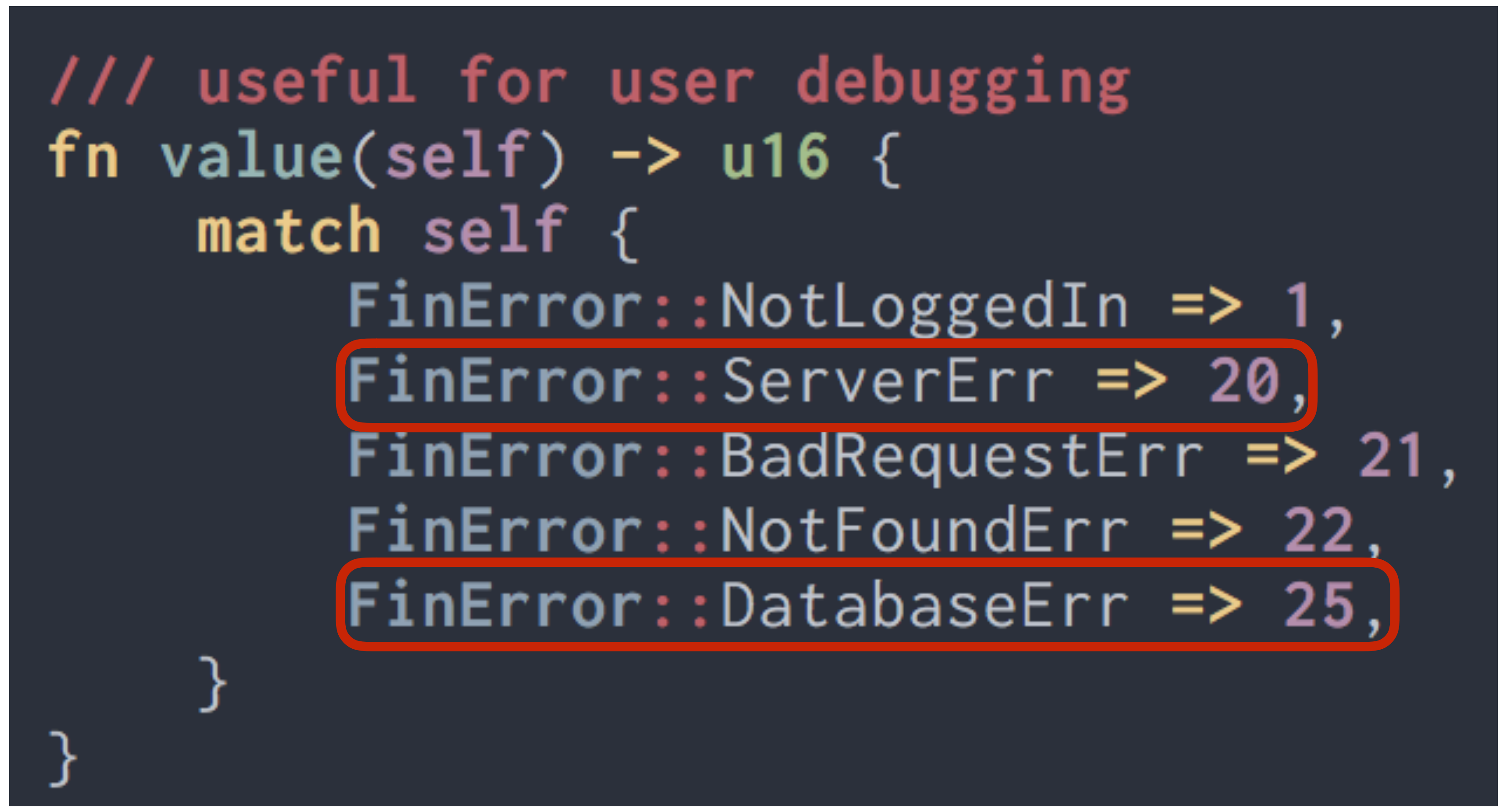
- declare a **user error** struct
  - code = info for developer
  - message = info for user



```rust
/// Return type to user
#[derive(Serialize)]
pub struct UserErrMessage {
    code: u16,
    message: String,
}
```



```rust
impl StdError for FinError {
    fn description(&self) -> &str {
        match self {
            FinError::NotLoggedIn => "user log-in required",
            FinError::BadRequestErr => "bad request",
            FinError::NotFoundErr => "not found",
            FinError::DatabaseErr | FinError::ServerErr => {
                "an error occured with the service"
            }
        }
    }
}
```



```rust
/// useful for user debugging
fn value(self) -> u16 {
    match self {
        FinError::NotLoggedIn => 1,
        FinError::ServerErr => 20,
        FinError::BadRequestErr => 21,
        FinError::NotFoundErr => 22,
        FinError::DatabaseErr => 25,
    }
}
```

Fin

Apoorv Kothari
toidiu.com
@toidiuCodes

github.com/toidiu/fin-public

auth

auth(cont..)

password management

- libpasta <https://libpasta.github.io/>
  - **Easy-to-use** password storage with **strong defaults** (scrypt).
    - *`libpasta::hash_password(&password);`*
    - *`libpasta::verify_password(&user.password_hash, &password)`*
  - **Migration support** for passwords to new algorithms.
    - `new_algo (old_algo ( password ))`

auth(cont..)

password management

- libpasta <https://libpasta.github.io/>
  - **Easy-to-use** password storage with **strong defaults** (scrypt).
    - *`libpasta::hash_password(&password);`*
    - *`libpasta::verify_password(&user.password_hash, &password)`*
  - **Migration support** for passwords to new algorithms.
    - `new_algo (`**`old_algo ( password )`**`)`

auth(cont..)

password management

- libpasta <https://libpasta.github.io/>
  - **Easy-to-use** password storage with **strong defaults** (scrypt).
    - `` `libpasta::hash_password(&password);` ``
    - `` `libpasta::verify_password(&user.password_hash, &password)` ``
  - **Migration support** for passwords to new algorithms.
    - `` `new_algo (old_algo ( password ))` ``

auth(cont..)

stateless session token

- paseto
  - paseto is JWT but with **sane defaults** and **smaller surface area**
    - you can specify `*version*` and `*purpose*`
    - only allows authenticated tokens

Fin

Apoorv Kothari

toidiu.com

@toidiuCodes

github.com/toidiu/fin-public