Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

ОТЧЕТ
по лабораторной работе №6
на тему

СОЗДАНИЕ ПРИЛОЖЕНИЯ ДЛЯ БАЗЫ ДАННЫХ

Студент: Кутняк А. В.

Руководитель: Игнатович А. О.

Минск 2024

1 ЦЕЛЬ

Цель лабораторной работы – создание прикладной программы для работы с базой данных и выполняющей заданные транзакции, а также реализовать механизм работы с базой данных (CRUD).

2 ХОД ВЫПОЛНЕНИЯ

Реализованное приложение – веб-сервис с интерфейсом обращения к базе данных через HTTP-запросы (PUT, GET, PATCH, DELETE).

2.1 Язык программирования

Для разработки проекта был выбран язык программирования Rust, предоставляющий инструменты для создания надежных и производительных приложений различной направленности.

2.2 Описание зависимостей

– Rocket.rs – web-фреймворк с асинхронным рантаймом;
– Diesel.rs – объектно-реляционная модель и др.;
– diesel-async – расширение Diesel с поддержкой асинхронного взаимодействия с Postgres и MySQL;
– rocket-db-pools – интеграция пула подключений асинхронного драйвера базы данных для Rocket.

Основным фактором в выборе зависимостей стала поддержка ими асинхронности, что позволяет реализовать полностью асинхронное взаимодействие с базой данных, обеспечивающее минимальные издержки на обслуживание множественных подключений к сервису.

2.3 Описание инструментов

– diesel-cli – выполняет генерацию миграций и декларативной схемы данных, необходимой при составлении моделей Diesel;
– diesel-cli-ext – предоставляет дополнительные возможности, например, автоматическую генерацию моделей на основе схемы БД.

Использование diesel-cli-ext обусловлено тривиальностью моделей таблиц, необходимых для реализации CRUD-запросов, а также объемом работы, которую требуется проделать для написания моделей вручную.

Для организации исходного кода проекта использована древовидная иерархическая система модулей:

```
src
|-- lib.rs
|-- main.rs
|-- models
|   |-- apartment.rs
|   |-- ...
|-- routes
|   |-- apartment.rs
|   |-- ...
|-- schema.rs
```

Модуль *lib.rs* содержит реализацию структуры *ApiResponse*, представляющую собой транспорт ответа на запрос в виде JSON-сообщения и статуса выполнения:

```
#[derive(Debug)]
pub struct ApiResponse {
    pub json: Value,
    pub status: Status,
}

impl ApiResponse {
    pub fn from(r: Result<impl Serialize, diesel::result::Error>) -> Self {
        match r {
            Ok(v) => Self::ok(v),
            Err(v) => Self::err(v),
        }
    }

    pub fn ok(v: impl Serialize) -> Self {...}
    pub fn err(v: diesel::result::Error) -> Self {...}
}

impl<'r> Responder<'r, 'r> for ApiResponse {
    fn respond_to(self, req: &Request) -> response::Result<'r> {
        Response::build_from(self.json.respond_to(req).unwrap())
            .status(self.status)
            .header(ContentType::JSON)
            .ok()
    }
}
```

Модуль *main.rs* определяет основные модули проекта, а также задает пул подключений к базе данных и точку входа Rocket.rs:

```
mod schema;
mod models;
mod routes;

use rocket::*;
use rocket_db_pools::Database;
use rocket_db_pools::diesel::PgPool;

#[derive(Database)]
#[database("hotel")]
pub struct MyDatabase(PgPool);

#[launch]
async fn rocket() -> _ {
    build()
        .attach(MyDatabase::init())
        .mount("/api", routes::api_routes())
}
```

Структура *MyDatabase* задает привязку пула асинхронных подключений Rocket к базе данных, параметры которого определяются через переменную среды ROCKET_DATABASES или через файл Rocket.toml:

```
// Rocket.toml
[global.databases]
hotel = { url = "postgres://postgres:pass@localhost/hotel" }
```

Модуль *schema.rs* содержит сгенерированную утилитой *diesel-cli* схему данных, используемую Diesel для связывания данных с моделями в коде:

```
// @generated automatically by Diesel CLI.

diesel::table! {
    apartment (no) {
        no -> Int4,
        free -> Bool,
        type_id -> Int4,
    }
}

...
```

Модуль *models.rs* аггрегирует структуры моделей данных, определенных в подмодулях каталога *models*; рассмотрим реализацию моделей данных на примере *apartment_reservation.rs*:

```
#[derive(Queryable, Identifiable, Serialize, Deserialize)]
#[diesel(table_name = apartment_reservation)]
pub struct ApartmentReservation {
    pub id: i32,
    pub date: NaiveDate,
    pub paid: bool,
    pub check_in: NaiveDate,
    pub check_out: NaiveDate,
    pub apartment_no: i32,
    pub customer_id: i32,
}

#[derive(Insertable, Deserialize)]
#[diesel(table_name = apartment_reservation)]
pub struct NewApartmentReservation {
    pub date: NaiveDate,
    pub paid: bool,
    pub check_in: NaiveDate,
    pub check_out: NaiveDate,
    pub apartment_no: i32,
    pub customer_id: i32,
}

#[derive(AsChangeset, Deserialize)]
#[diesel(table_name = apartment_reservation)]
pub struct UpdateApartmentReservation {
    pub date: Option<NaiveDate>,
    pub paid: Option<bool>,
    pub check_in: Option<NaiveDate>,
    pub check_out: Option<NaiveDate>,
    pub apartment_no: Option<i32>,
    pub customer_id: Option<i32>,
}
```

Сгенерированные утилитой *diesel-cli-ext* модели были дополнены структурами New- и Update- моделями для использования в качестве аргументов для запросов PUT и PATCH соответственно.

Реализация дополнительных моделей может показаться излишней, однако в реальных задачах данные модели значительно различаются.

Модуль *routes.rs* передает Rocket набор обработчиков запросов, реализованных в подмодулях соответствующего каталога:

```
pub fn api_routes() -> Vec<Route> {
    routes![
        apartment::create,
        apartment::list, apartment::read,
        apartment::update, apartment::delete,
        ...
    ]
}
```

Как и в случае с моделями, реализации однотипных запросов для разных моделей являются тривиальными и в основном повторяют друг друга, поэтому рассмотрим их на примере *apartment.rs*:

```
#[post("/apartment", format="application/json", data="<apartment>")]
pub async fn create(mut db: Connection<MyDatabase>,
                    apartment: Json<NewApartment>) -> ApiResponse {
    let response =
        diesel::insert_into(apartment::table)
        .values(&apartment.into_inner())
        .get_result::<Apartment>(&mut db).await;

    ApiResponse::from(response)
}

#[get("/apartments")]
pub async fn list(mut db: Connection<MyDatabase>) -> ApiResponse {
    let response =
        apartment::table
        .select(apartment::all_columns).load::<Apartment>(&mut db).await;

    ApiResponse::from(response)
}

#[get("/apartment/<no>")]
pub async fn read(mut db: Connection<MyDatabase>, no: i32) -> ApiResponse {
    let response =
        apartment::table
        .find(no).first::<Apartment>(&mut db).await;

    ApiResponse::from(response)
}
```

```
#[patch("/apartment/<no>", format="application/json", data="<apartment>")]
pub async fn update(mut db: Connection<MyDatabase>,
                    apartment: Json<UpdateApartment>,
                    no: i32) -> ApiResponse {
    let response =
        diesel::update(apartment::table.find(no))
        .set(&apartment.into_inner()).execute(&mut db).await;

    match response {
        Ok(count) => ApiResponse::ok(json!({"update": count})),
        Err(err) => ApiResponse::err(err),
    }
}

#[delete("/apartment/<no>")]
pub async fn delete(mut db: Connection<MyDatabase>,
                    no: i32) -> ApiResponse {
    let response
        = diesel::delete(apartment::table.find(no)).execute(&mut db).await;

    match response {
        Ok(count) => ApiResponse::ok(json!({"delete": count})),
        Err(err) => ApiResponse::err(err),
    }
}
```

Следует отметить важный момент: язык Rust задает «тренд» статических проверок типов на этапе компиляции; на примере Diesel + Rocket это означает следующее:

1. Diesel сопоставляет сгенерированную схему данных с прописанными в коде моделями данных;

2. Rocket сопоставляет принимаемые аргументы с декларируемыми с структуре маршрута аргументами запроса;

3. Успешная компиляция возможна лишь в случае согласованности схемы данных с моделями данных, а также правильности принимаемых обработчиками аргументов, что исключает неверную интерпретацию данных на этапе исполнения;

4. Помимо этого, все, что выполняется на этапе компиляции, остается на этапе компиляции; для нас это значит одно - абстракции будут поняты и применены с наименьшей стоимостью реализации.

3 ДЕМОНСТРАЦИЯ РАБОТЫ ПРИЛОЖЕНИЯ

Ниже в формате «запрос-ответ» представлен результат выполнения запросов, выполненных при помощи утилиты *curl*:

```
> curl --location 'http://127.0.0.1:8000/api/apartment/111'
{"error":"Record not found"}%

> curl --location 'http://127.0.0.1:8000/api/apartments'
[{"free":false,"no":201,"type_id":2},
 {"free":false,"no":302,"type_id":3},
 {"free":true,"no":402,"type_id":4},
 {"free":false,"no":502,"type_id":5},
 {"free":true,"no":603,"type_id":6},
 {"free":true,"no":101,"type_id":1}]%

> curl --location 'http://localhost:8000/api/apartment' \
--header 'Content-Type: application/json' \
--data '{"no" : 111, "free": false, "type_id" : 1}'
{"free":false,"no":111,"type_id":1}%

> curl --location 'http://127.0.0.1:8000/api/apartment/111'
{"free":false,"no":111,"type_id":1}%

> curl --location --request PATCH 'http://localhost:8000/api/apartment/111' \
--header 'Content-Type: application/json' \
--data '{ "free": true }'
{"update":1}%

> curl --location 'http://127.0.0.1:8000/api/apartment/111'
{"free":true,"no":111,"type_id":1}%

> curl --location 'http://localhost:8000/api/apartment' \
--header 'Content-Type: application/json' \
--data '{"no" : 111, "free": false, "type_id" : 1}'
{"error":"duplicate key value violates unique constraint \"apartment_pkey\""}%

> curl --location --request DELETE 'http://localhost:8000/api/apartment/111'
{"delete":1}%

> curl --location --request DELETE 'http://localhost:8000/api/apartment/111'
{"delete":0}%
```

При запуске артефакта приложения, собранного под профилем *debug*, рантайм Rocket выводит различную информацию, такую как сведения о конфигурации веб-сервиса, список зарегистрированных маршрутов, а также логи исполнения тех или иных обработчиков:

```
📡 Fairings:
   >> 'hotel' Database Pool (ignite, shutdown)
   >> Shield (liftoff, response, singleton)
🛡 Shield:
   >> X-Content-Type-Options: nosniff
   >> X-Frame-Options: SAMEORIGIN
   >> Permissions-Policy: interest-cohort=()
🚀 Rocket has launched from http://127.0.0.1:8000
POST /api/apartment application/json:
   >> Matched: (create) POST /api/apartment application/json
   >> Outcome: Success(200 OK)
   >> Response succeeded.
GET /api/apartments:
   >> Matched: (list) GET /api/apartments
   >> Outcome: Success(200 OK)
   >> Response succeeded.
GET /api/apartment/111:
   >> Matched: (read) GET /api/apartment/<no>
   >> Outcome: Success(200 OK)
   >> Response succeeded.
PATCH /api/apartment/111 application/json:
   >> Matched: (update) PATCH /api/apartment/<no> application/json
   >> Outcome: Success(200 OK)
   >> Response succeeded.
DELETE /api/apartment/111:
   >> Matched: (delete) DELETE /api/apartment/<no>
   >> Outcome: Success(200 OK)
   >> Response succeeded.
DELETE /api/apartment/111:
   >> Matched: (delete) DELETE /api/apartment/<no>
   >> Outcome: Success(200 OK)
   >> Response succeeded.
```

Рисунок 3.1 – Отладочные сообщения Rocket.rs