Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем і технологій

Лабораторна робота №5

з дисципліни «Основи WEB-технологій»

Тема: «GraphQL, Створення Schema GraphQL та Resolvers. Створення Query та Mutation»

Виконав:
студент групи ІА-11
Воробей Антон Олегович

Перевірив:
Альбрехт Й. О.

Київ 2024

Завдання:

- На своїй БД (розробленої в лаб. роб. #4) за допомогою Schema Definition Language (SDL) створити схему GraphQL.
- Додати Resolvers для виконання операцій GraphQL. • Створити та виконати Query та Mutation для виконання операцій додавання, редагування та видалення інформації (CRUD) в БД.
- Виконати дослідження роботи створених query та mutation за допомогою Postman.

Посилання на вихідний код проекта:

https://github.com/tonujet/data-in-cloud/tree/feature/lab5

Хід роботи

Дану лабораторну роботу я буду реалізовувати за допомогою мови програмування rust. Як бібліотеку для побудови backend застосунків було використано axum. Для створення GraphQL schema та відповідних GraphQL resolvers я використаю бібліотеку async_graphql, що дозволяє використовувати programming language first approach. Тобто мені не потрібно створювати GraphQL schema своїми руками - вона буде згенерована у відповідності до всіх анотацій, що я розставлю в коді

Заінітимо GraphQL

```
pub fn api_routes(state: AppState) -> Router {
    let schema = schema(state.clone());
    let api_router = Router::new()
        .route("/graphql", get(graphiql).post_service(GraphQL::new(schema)))
        .nest(EntityApi::Repos.to_endpoint(),
rest_repo_controller::routes(state.clone()))
        .nest(EntityApi::Users.to_endpoint(),
rest_user_controller::routes(state.clone()))
        .nest(EntityApi::Users.to_endpoint(),
user_repo_controller::routes(state.clone()))
        .nest(
            EntityApi::UserRepoInfos.to_endpoint(),
            rest_user_repo_info_controller::routes(state.clone()),
        );

    Router::new()
        .nest(API_PREFIX.as_str(), api_router)
}
```

Розставимо анотації що будуть описувати сутності

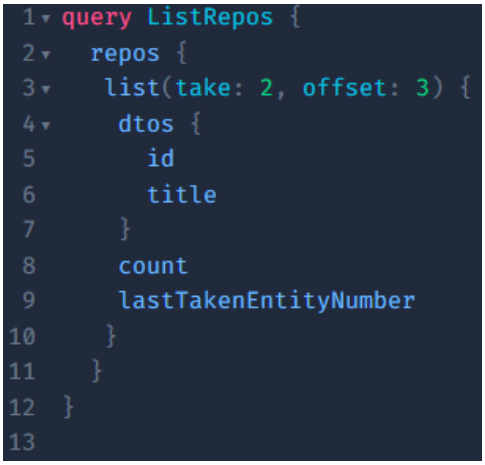
```
#[derive(Serialize, Deserialize, Debug, Eq, Clone)]
#[derive(async_graphql::SimpleObject)]
pub struct RepoDto {
    pub id: Uuid,
    pub title: String,
    pub description: Option<String>,
    pub repo_type: RepositoryType,
    pub stars: u64,
    pub created: chrono::NaiveDateTime,
    pub updated: chrono::NaiveDateTime,
}
```

Створимо відповідні GraphQL resolvers

```
#[derive(Default)]
pub struct QueryRepo;

#[Object]
impl QueryRepo {
    async fn get<'a>(&self, ctx: &Context<'a>, id: Uuid) ->
async_graphql::Result<RepoDto> {
        let AppState {
            repo_state: state, ..
        } = ctx.data_unchecked::<AppState>();
        ctx.data_unchecked::<AppState>();
        state.service.get(&id).await.extend()
    }

    async fn list<'a>(
        &self,
        ctx: &Context<'a>,
        take: Option<u64>,
        offset: Option<u64>,
    ) -> async_graphql::Result<GraphqlDtoList<RepoDto>> {
        let AppState {
            repo_state: state, ..
        } = ctx.data_unchecked::<AppState>();
        ctx.data_unchecked::<AppState>();
        state
            .service
            .list(take, offset)
            .await
            .map(|dto_list| dto_list.into())
            .extend()
    }
}

#[derive(Default)]
pub struct MutationRepo;

#[Object]
impl MutationRepo {
    async fn delete<'a>(&self, ctx: &Context<'a>, id: Uuid) ->
async_graphql::Result<RepoDto> {
        let AppState {
            repo_state: state, ..
        } = ctx.data_unchecked::<AppState>();
        ctx.data_unchecked::<AppState>();
        state.service.delete(&id).await.extend()
    }
```

```rust
    async fn create<'a>(
        &self,
        ctx: &Context<'a>,
        #[graphql(validator(custom = "GraphQLValidator::default()"))] repo_dto:
CreateUpdateRepoDto,
    ) -> async_graphql::Result<RepoDto> {
        let AppState {
            repo_state: state, ..
        } = ctx.data_unchecked::<AppState>();
        ctx.data_unchecked::<AppState>();
        state.service.create(repo_dto).await.extend()
    }

    async fn update<'a>(
        &self,
        ctx: &Context<'a>,
        id: Uuid,
        #[graphql(validator(custom = "GraphQLValidator::default()"))] repo_dto:
CreateUpdateRepoDto,
    ) -> async_graphql::Result<RepoDto> {
        let AppState {
            repo_state: state, ..
        } = ctx.data_unchecked::<AppState>();
        ctx.data_unchecked::<AppState>();
        state.service.update(&id, repo_dto).await.extend()
    }
}
```

Перевіримо роботу застосунку на звичайних CRUD операціях

Запит	Відповідь
	
	

```
mutation UpdateRepo {
  repos {
    update(
      id: "16307316-3ec7-44cc-ba8e-b104c6520e38"
      repoDto: {
        title: "Updated desc"
        description: "Updated desc"
        repoType: PRIVATE
      }
    ) {
      id
      title
      description
    }
  }
}
```

```
{
  "data": {
    "repos": {
      "update": {
        "id": "16307316-3ec7-44cc-ba8e-b104c6520e38",
        "title": "Updated desc",
        "description": "Updated desc"
      }
    }
  }
}
```

```
query GetRepo1 {
  repos {
    get(
      id: "16307316-3ec7-44cc-ba8e-b104c6520e38"
    ) {
      id
      title
    }
  }
}
```

```
{
  "data": {
    "repos": {
      "get": {
        "id": "16307316-3ec7-44cc-ba8e-b104c6520e38",
        "title": "Updated desc"
      }
    }
  }
}
```

```
mutation DeleteRepo {
  repos {
    delete(
      id: "16307316-3ec7-44cc-ba8e-b104c6520e38"
    ) {
      id
      title
    }
  }
}
```

```
{
  "data": {
    "repos": {
      "delete": {
        "id": "16307316-3ec7-44cc-ba8e-b104c6520e38",
        "title": "Updated desc"
      }
    }
  }
}
```

```
query GetRepo2 {
  repos {
    get(
      id: "5bb82216-0c9c-49f0-9f69-6e5945869d6f"
    ) {
      id
      title
    }
  }
}
```

```
{
  "data": null,
  "errors": [
    {
      "message": "Repository with uuid 5bb82216-0c9c-49f0-9f69-6e5945869d6f was deleted",
      "locations": [
        {
          "line": 68,
          "column": 4
        }
      ],
      "path": [
        "repos",
        "get"
      ],
      "extensions": {
        "error_name": "RepositoryError"
      }
    }
  ]
}
```

Висновок: На цій лабораторні роботі я дослідив, що таке GraphQL. На практиці я побудував API мовою rust, до якого всі звернення відбуваються саме через GraphQL