Employee API Documentation

Requirements:

- Go version v1.20. For more detailed information related to Go installation, you can open this link.
- Docker engine version 19.03.0+ because I use docker-compose version 3.8. For more detailed information related to Docker installation, you can open this link.

Task 1: Database Setup

We use Dockerfile to setup PostgreSQL database. This task/step only will create a standalone PostgreSQL database docker image and container (not connected to Go API container because it will on the Task 5 explanation). There are 2 options to create employees table and its data:

- Use Dockerfile command using init.sql

  This option copy init.sql file which contains all SQL statements to populate employees table and its data to the container. We only need to build the PostgreSQL docker image and create the container. Run these commands:

  - docker build -t {your-postgresdb-image-name} -f postgresdb.dockerfile .

    It will create a new PostgreSQL docker image.
  - docker container run -d --name {your-postgresdb-container-name} -p 5432:5432 {your-postgresdb-image-name}

    It will create a new PostgreSQL docker container with exposed port 5432 (you can customize the port too).

  If there is no error, you can use the database for any needs (e.g. Employee API) and you can manage the database using database management tool (e.g. Dbeaver or Datagrip).

Before run the above docker commands, you can change some env of PostgreSQL in **postgresdb.dockerfile** based on your needs (Db name, username, and password).

- Use Employee API's migration feature

  Basically this option still use Dockerfile to create PostgreSQL docker image and container. The difference from the other option is how to create and populate employees table and data. Therefore, we need to build and create PostgreSQL docker image and container using those 2 commands which is in the previous option but we must to comment the `COPY migrations/seeds/init.sql /docker-entrypoint-initdb.d` command so it does not create and populate employees table and data after PostgreSQL container is created.

  After PostgreSQL container successfully run without error, we can start to create and populate employees table and its data using migration feature. You can use these commands:

  - `go run main.go migratenew {your_migration_file_name}`
    The command will create new migration file in migrations/sql directory.
  - `go run main.go migrate`
    The command will migrate up all available database migrations since the latest migration in the database.
  - `go run main.go migratedown`
    The command will migrate down only one or the latest database migration.

  Before run the docker commands build and container run, you can change some env of PostgreSQL in **postgresdb.dockerfile** based on your needs (Db name, username, and password).

Task 2: Golang RESTful API

There are 5 endpoints in this API:

- GET /employees
    - Request Body: -
    - Response Body:


```
{
   "status_code": 200,
   "status": "success",
   "data": [
      {
         "id": 1,
         "first_name": "Ajax",
         "last_name": "Zaher",
         "email": "Ajax_Zaher@email.com",
         "hire_date": "2023-02-12T00:00:00Z"
      },
      {
         "id": 2,
         "first_name": "Djedefre",
         "last_name": "Narantsetseg",
         "email": "Djedefre_Narantsetseg@email.com",
         "hire_date": "2024-01-03T00:00:00Z"
      }
   ]
}
```

- GET /employees/{id}
    - Request Body: -
    - Response Body: Assume we pass id = 5


```
{
   "status_code": 200,
   "status": "success",
   "data": [
      {
         "id": 5,
         "first_name": "Shaindel",
         "last_name": "Garbine",
         "email": "Shaindel_Garbine@email.com",
         "hire_date": "2019-07-17T00:00:00Z"
      }
   ]
}
```


- POST /employees
    - Request Body:


```
{
   "first_name": "John",
   "last_name": "Doe",
   "email": "john_doe32@email.com",
   "hire_date": "2012-03-21"
}
```


Assumption: All fields required to be filled in request body with first_name & last_name fields need to be alphanum string type, email is email string type and hire_date need to be valid date string (yyyy-mm–dd format).

- Response Body:

```
{
   "status_code": 200,
   "status": "success",
   "data": [
      {
         "id": 11,
         "first_name": "John",
         "last_name": "Doe",
         "email": "john_doe32@email.com",
         "hire_date": "2012-03-21T00:00:00Z"
      }
   ]
}
```

- PUT /employees/{id}
  - Request Body:

```
{
   "first_name": "Mike",
   "last_name": "Wazowski",
   "email": "mike_wazowski10@email.com",
   "hire_date": "2014-10-03"
}
```

Assumption: All fields required to be filled in request body with first_name & last_name fields need to be alphanum string type, email is email string type and hire_date need to be valid date string (yyyy-mm–dd format).

- Response Body: Assume we pass id = 1


```
{
   "status_code": 200,
   "status": "success",
   "data": [
      {
         "id": 1,
         "first_name": "Mike",
         "last_name": "Wazowski",
         "email": "mike_wazowski10@email.com",
         "hire_date": "2014-10-03T00:00:00Z"
      }
   ]
}
```


- DELETE /employees{id}
  - Request Body : -
  - Response Body: Assume we pass id = 11


```
{
   "status_code": 200,
   "status": "success",
   "data": []
}
```

Notes:

- I added **status_code** and **status** field in every response body to make it more readable.
- For error cases, I injected the error with some message so the real error from the API only showed in log not in response body. So far, we have 3 error cases, 500, 404, and 400. For the example, the API's default error is 500 (internal server error) and will be like this:

```
{
   "status_code": 500,
   "status": "error",
      "message": "Ups, something wrong with the server. Please try again later"
}
```

- For more detailed information related API spec, you can export Postman collections from **docs/postman/docs/postman/Employee API.postman_collection.json** file to your Postman application.

Task 3: Unit Testing

Strategy:

- I use stretchr/testify library to help unit test preparation and to compare the expected output with actual output using assert package.
- For mocking, I use gomock to mock method in service and repository layer. For call to database mock, I use sqlmock.
- I only made unit test for endpoint Get All Employees, Get Employee by ID, and some of Create Employee case.
- I try to test every case (success & error) in every methods by comparing the expected output with actual output (returned data and error) except for the handler layer, I only check the status code.
- You can use command `go test -coverprofile=coverage.out ./... ; go tool cover -func=coverage.out` to run the unit test and display the unit test coverage result.

Task 4 & 5: Documentation & Docker-Compose

For this part, I assume you guys already setting up the Docker and Go.

- Run API server in local

  Before you run the API server, you need to create **.env** file by copying **.env.sample** files and fill the env values based on your needs. Then you need to setup the PostgreSQL database first. You can use your local PostgreSQL database or just follow task 1 explanation to setup PostgreSQL docker container. You can use one of these commands to run the API server in local:

  - Go run main.go
  - Go build && ./mid-developer-test

- Run connected PostgreSQL & API server

  This point can be done by using **docker-compose.yml** file located in root folder of this project. For PostgreSQL database configuration in docker-compose, you can set the environment like (db name, username, password) in file **postgresdb.dockerfile** and set the port in docker-compose file. For API config, you can just edit the env value in **.env** file because when the API image is being built by docker, it will copy the **.env** file value to the container. If you already done with the configuration, you can run this command to build the docker images and run the containers docker-compose up -d --build.