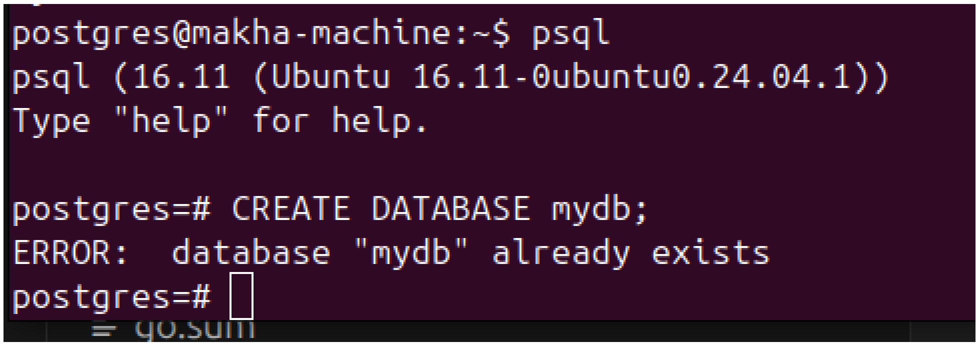
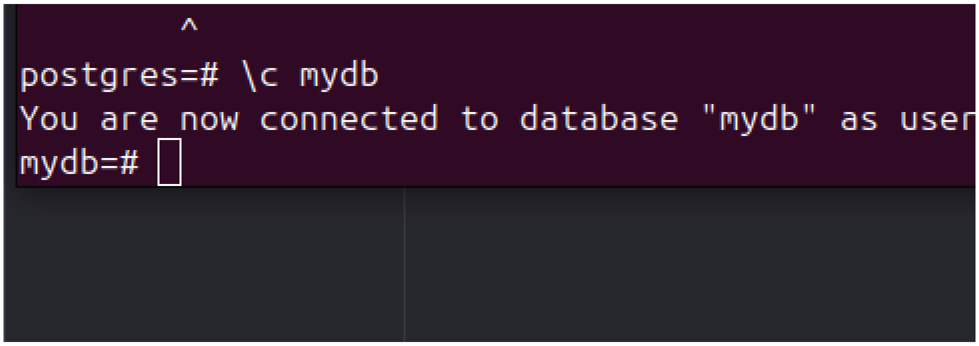
Step 1: `Create database`

Mine already exists so I have the following output:



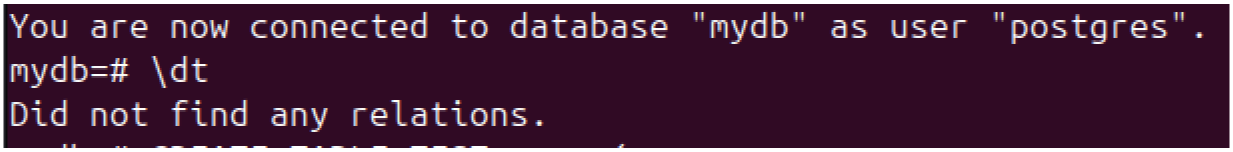
```
postgres@makha-machine:~$ psql
psql (16.11 (Ubuntu 16.11-0ubuntu0.24.04.1))
Type "help" for help.

postgres=# CREATE DATABASE mydb;
ERROR:  database "mydb" already exists
postgres=# 
    = go.sum
```

1.1 Connect to your database:



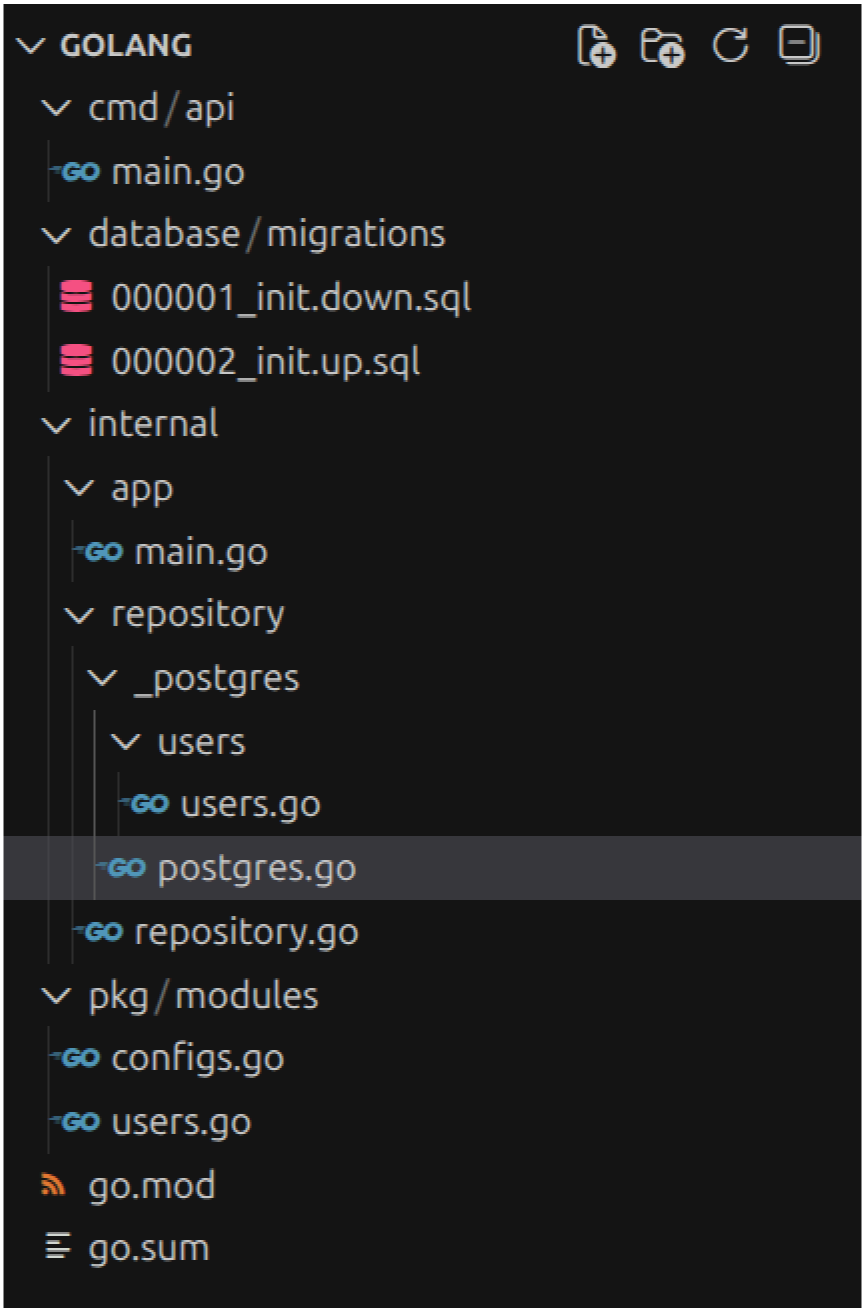
```
          ^
postgres=# \c mydb
You are now connected to database "mydb" as user
mydb=# 
```

1.2 Make sure there are no objects



```
You are now connected to database "mydb" as user "postgres".
mydb=# \dt
Did not find any relations.
```

Step 2: `Define project structure:`



```
GOLANG
  cmd / api
    main.go
  database / migrations
    000001_init.down.sql
    000002_init.up.sql
  internal
    app
      main.go
    repository
      _postgres
        users
          users.go
        postgres.go
      repository.go
  pkg / modules
    configs.go
    users.go
  go.mod
  go.sum
```

Step 3: Define function to connect to DB

Create migration scripts in the `database/migrations` directory in the root of the project:

000001_init.down.sql

```
drop table if exists users cascade;
```

000002_init.up.sql

```sql
create table if not exists users (
   id serial primary key,
   name varchar(255) not null
);

insert into users (name) values ('John Doe');
```

In the `internal/repository/_postgres/postgres.go` type:

```go
package _postgres

import (
	"context"
	"fmt"
	"github.com/jmoiron/sqlx"
	"github.com/golang-migrate/migrate/v4"
	"golang/pkg/modules"
	_ "github.com/lib/pq"
	_ "github.com/golang-migrate/migrate/v4/database/postgres"
	_ "github.com/golang-migrate/migrate/v4/source/file"
)

type Dialect struct {
	DB *sqlx.DB
}

func NewPGXDialect(ctx context.Context, cfg *modules.PostgreConfig) *Dialect {
	dsn := fmt.Sprintf("host=%s port=%s user=%s password=%s dbname=%s sslmode=%s",
		cfg.Host, cfg.Port, cfg.Username, cfg.Password, cfg.DBName, cfg.SSLMode)

	db, err := sqlx.Connect("postgres", dsn)

	if err != nil {
		panic(err)
	}

	err = db.Ping()

	if err != nil {
		panic(err)
	}
	AutoMigrate(cfg)

	return &Dialect{
		DB: db,
	}
}
```

```
func AutoMigrate(cfg *modules.PostgreConfig) {
	sourceURL := "file://database/migrations"
	databaseURL := fmt.Sprintf("postgres://%s:%s@%s:%s/%s?sslmode=%s",
		cfg.Username, cfg.Password, cfg.Host, cfg.Port, cfg.DBName,
cfg.SSLMode)

	m, err := migrate.New(sourceURL, databaseURL)

	if err != nil {
		panic(err)
	}

	err = m.Up()

	if err != nil && err != migrate.ErrNoChange {
		panic(err)
	}
}
```

Step 4 : Define `Configs`

In the `pkg/modules/configs.go` define the following structure:

```
package modules

import (
	"time"
)

type PostgreConfig struct {
	Host string
	Port string
	Username string
	Password string
	DBName string
	SSLMode string
```

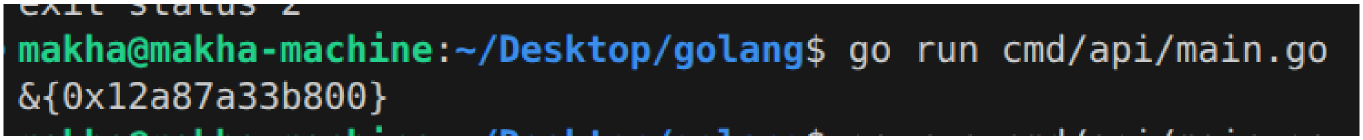
```
    ExecTimeout time.Duration
}
```

Step 5: Test the migration functionality

In the `internal/app/main.go` run your `NewPGXDialect` function:

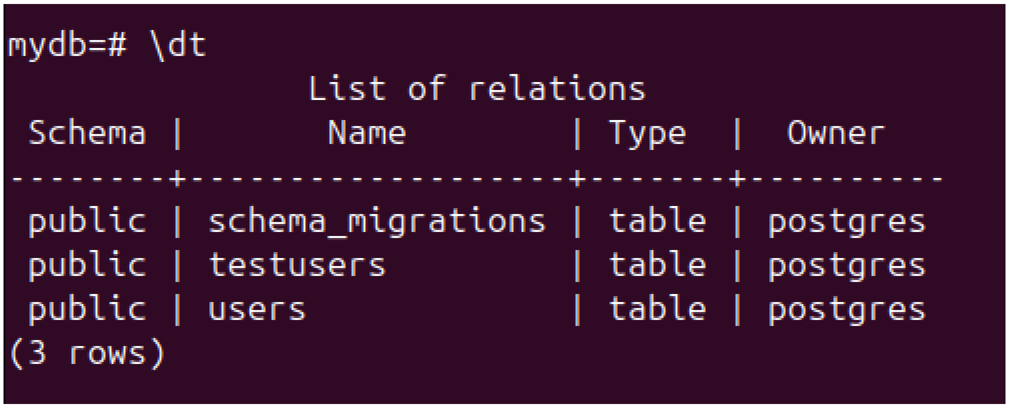
```go
package app

import(
....
)

func Run(){
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    dbConfig := initPostgreConfig()

    _postgre := _postgres.NewPGXDialect(ctx, dbConfig)

    fmt.Println(_postgre)
}

func initPostgreConfig() *modules.PostgreConfig {
    return &modules.PostgreConfig{
        Host: "localhost",
        Port: "5432",
        Username: "postgres",
        Password: "postgres",
        DBName: "mydb",
        SSLMode: "disable",
        ExecTimeout: 5 * time.Second,
    }
}
```

You should get something like that:



```
exit status 2
makha@makha-machine:~/Desktop/golang$ go run cmd/api/main.go
&{0x12a87a33b800}
```

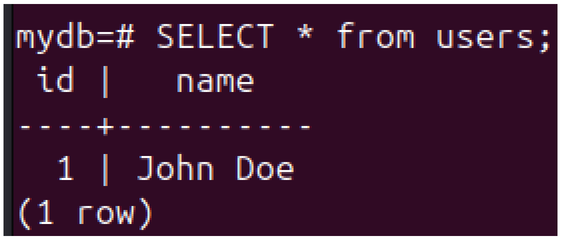
Let's return to our database we've created in the step 1:



```
mydb=# \dt
               List of relations
 Schema |        Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | schema_migrations | table | postgres
 public | testusers         | table | postgres
 public | users             | table | postgres
(3 rows)
```

Now, we have `users` table. Do not pay attention to `testusers`.

Let's try to query this table:



```
mydb=# SELECT * from users;
 id |   name
----+----------
  1 | John Doe
(1 row)
```

> Nice work!

Step 6: Define the `User` struct:

In the `pkg/modules/users.go` define `User`

```
package modules

type User struct {
	ID   int    `db:"id"`
	Name string `db:"name"`
}
```

```
> Good!
```

Step 7: Create `UserRepository`

Define the struct and initializer func in the `internal/repository/_postgres/users/users.go`:

```
package users
```

```
import (
    ...
)

type Repository struct {
    db *_postgres.Dialect
    executionTimeout time.Duration
}

func NewUserRepository(db *_postgres.Dialect) *Repository {
    return &Repository{
        db: db,
        executionTimeout: time.Second * 5,
    }
}

func (r *Repository) GetUsers() ([]modules.User, error) {
    var users []modules.User
    err := r.db.DB.Select(&users, "SELECT id, name FROM users")
    if err != nil {
        return nil, err
    }

    fmt.Println(users)
    return users, nil
}
```

In the internal/_repository/repository.go define `UserRepository` interface, `Repositories` Struct, and `NewRepositories` func.

```
package repository

import (
...
)

type UserRepository interface {
    GetUsers() ([]modules.User, error)
}

type Repositories struct {
    UserRepository
}
```

```
func NewRepositories(db *_postgres.Dialect) *Repositories {
	return &Repositories{
		UserRepository: users.NewUserRepository(db),
	}
}
```

> Nice!

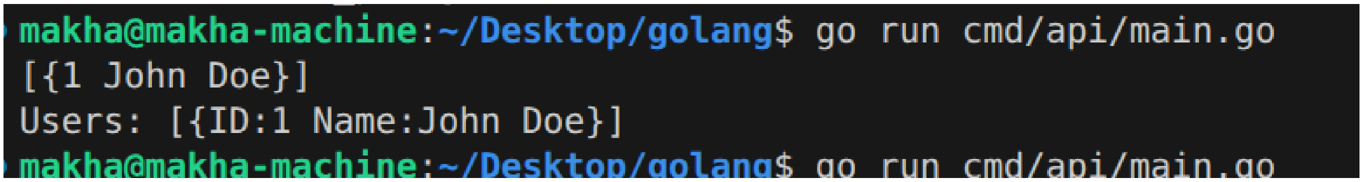
Step 8: Implement the functionality

In the `internal/app/main.go`:

```
package app

import(
	...
)

func Run(){
	ctx, cancel := context.WithCancel(context.Background())
	defer cancel()

	dbConfig := initPostgreConfig()

	_postgre := _postgres.NewPGXDialect(ctx, dbConfig)

	repositories := repository.NewRepositories(_postgre)

	users, err := repositories.GetUsers()
	if err != nil {
		fmt.Printf("Error fetching users: %v\n", err)
		return
	}

	fmt.Printf("Users: %+v\n", users)
}

func initPostgreConfig() *modules.PostgreConfig {
	return &modules.PostgreConfig{
		Host: "localhost",
		Port: "5432",
		Username: "postgres",
		Password: "postgres",
        DBName: "mydb",
        SSLMode: "disable",
        ExecTimeout: 5 * time.Second,
    }
}
```

Step 9: Run this code:

If you'll run this code you will get something like this:



```
makha@makha-machine:~/Desktop/golang$ go run cmd/api/main.go
[{1 John Doe}]
Users:  [{ID:1 Name:John Doe}]
makha@makha-machine:~/Desktop/golang$ go run cmd/api/main.go
```

> Congrats! You've just got the information from your `database`, `parsed it` to struct and returned to the console!

Individual task(3 points):

1. Expanging the `UserRepository`:

- **Create a NewUser:**
    - **Expand the `User` struct and table by adding 3 more fields and columns respectively.**
    - **Learn how to handle `Insert` statements and return the newly generated ID.**
    - **Requirement: Handle potential error cases.**
    - **Deliverables: User created successfully and code handles potential error cases.**
- **Update an Existing User:**
    - **Learn how to `Update` existing data and handle `RowsAffected`.**
    - **Requirement: Add a check to see if the user actually existed. If `0` rows were affected, return a custom informative error.**
    - **Deliverables: User updated successfully and code handles potential error cases.**
- **Get `User` by `ID`:**
    - **Practice fetching a single record from DB**

    - **Requirement:** Handle the case where the ID doesn't exist by returning a `nil` user and a clear informative error message.
    - **Deliverables:** Correct User fetched by ID, all potential error cases are handled.
- Delete `User` by `ID`:
    - Practice deleting a record from DB
    - **Requirement:** Handle the case where the ID doesn't exist, return the rows affected if possible.
    - **Deliverables:** User Deleted successfully and potential error cases are handled

2. Expose handler function for each of the functions of the `UserRepository`

Connect the `Hadlers` layer with the `Usecase` layer. Then, connect `Usecase` layer with the `Repository` layer by simply redirecting the function call from `Handler` layer to the `Repository` layer.

Example(pseudocode):

```
func (u *UserUsecase) CreateUser(name string) string {
     response, err := u.repo.CreateUser(name)
     return fmt.Sprintf("%v", response)
}
```

`Deliverables`:

- At least 5 endpoints were exposed: GET, GET/{id}, POST, PATCH/PUT, DELETE (90%)
- All of the necessary error handling were done on the handlers level (10%)

3. Add logging and authentication middleware

- Log all the http responses and requests made in our service.
    - `timestamp`, `http method`, and `endpoint name` are required in your log structure.

  - **Usage of standard golang log package is required. Logging with `fmt` package is prohibited.**
- **Check, whether user have provided the vaid "X-API-KEY" header.**
  - **If `X-API-KEY` header is missing or invalid then return `401-Unauthorized`.**
  - **If valid -> proceeds to handler.**

Definition of Done:

- `go run cmd/api/main.go` **starts the server on** `:8080`
- **All endpoints:**
  - **Returns JSON only**
  - **Uses correct HTTP status codes**
  - **Set** `Content-Type: application/json`
- `UserRepository`**:**
  - **Has at least** `5` **functions to:**
    - `GET all users`
    - `GET user by ID`
    - `CREATE new USER`
    - `UPDATE USER`
    - `DELETE USER`
- `Handler`
  - calls the `Usecase` and `Usecase` calls the `Repository` through interfaces
  - `Healthcheck` endpoint implemented.
- `Middleware`**:**
  - `Blocks` **requests without a valid API KEY**
  - **Logs every request**

Deliverables:

- **Push all the changes to your GitHub repository**
- **Submit the link to this repository**
- **Submit [1-2]-minute demo video of your project. You can speed it up if it is too long.**

> Below Are Marking Criteria

Criteria	Weight
The project and the whole flow runs successfully	0.5 pts
Student have written at least 5 endpoints for GET, GET{ID}, POST, PUT, DELETE methods and healthcheck endpoint	0.5 pts
User Repository has at least 5 functions for all CRUD operations	1 pts
Handler layer calls usecase layer, usecase layer calls repository layer	0.5 pts
Middlewares logs every request and checks the X-API-KEY header	0.5 pts
OVERALL	`3pts`

DEADLINE: 22.02.2026, SUNDAY 23:59

OPTIONAL FEATURES (Choose Freely)

Students may implement any number of the following to increase difficulty and engagement.

🐣 EASY: "The Baby Gopher"

- **Config via `.env` and/or `.yml` files: Use library like `godotenv` to load your `PostgreConfig` from a `.env` and/or `.yaml` instead of hardcoding it.**
- **API Documentation: Add API documentation with the `/swagger` endpoint.**

👷 MEDIUM: "The Gopher-at-Work"

- **Soft Deletes: Instead of deleting a row, add a `deleted_at` (timestamp) column. Update your `GetUsers` query to only return rows where `deleted_at IS NULL`**
- **Transaction Support: Implement a function where you create a User and an "Audit Log" entry at the same time using a Database Transaction (`db.Begin()`).**
- **Pagination: Add `limit` and `offset` parameters to your `GET /users` endpoint to handle large datasets.**

🧙 ADVANCED: "The Gopher Wizard"

- **Unit Testing with Mocks:** Write unit tests for your `Usecase` layer by creating a "Mock Repository." Use a tool like `mockery` or write the mock manually.
- **Graceful Shutdown:** Implement a listener that catches `SIGINT` or `SIGTERM` and closes the database connections and server properly before the process exits.
- **Password Hashing:** Add a `password` field to the User. Use `golang.org/x/crypto/bcrypt` to hash the password before saving it to the database.
- **Authentication Flow**: Implement Full `Bearer/Basic` Authentication/authorization flow with access tokens to provide authorized protected role-based access for every user.

🦾 EXTRA: "The Gopher Overlord"

- **Dockerization:** Provide a `docker-compose.yml` file that spins up both your Go Application and a PostgreSQL container with a single command.
- **Redis Caching:** Implement a caching layer in the Usecase. When `GetUserByID` is called, check **Redis** first. If not found, get it from Postgres and save it to Redis.
- **Background Workers:** Use a Goroutine and a `time.Ticker` to create a background task that prints the total number of users in the database to the console every 60 seconds.