Assignment 2, Go

completed by: Amira Ordiyeva, 22B22B1562

Exercise 1: Connecting to PostgreSQL Directly with database/sql

Objective: Connect to a PostgreSQL database, create a table, insert some records, and query them.

1. **Setup PostgreSQL Connection:** Create a Go program that connects to your PostgreSQL database using the `pq` driver.

```
amiraordiyeva=# CREATE DATABASE golang;
CREATE DATABASE
```

Я создала новый проект Go и инициализировал его с помощью команды:

```
⊗ amiraordiyeva@MacBook-Air-Amira golang % go mod init myproject

  go: /Users/amiraordiyeva/Desktop/golang/go.mod already exists
```

Затем я установила драйвер PostgreSQL для Go (pq) с помощью команды:

```
● amiraordiyeva@MacBook-Air-Amira golang % go get github.com/lib/pq
```

2. **Create a Table:** Write a function to create a simple table `users` with columns for `id`, `name`, and `age`.

```go
func createTable(db *sql.DB) {
    query := `CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL,
        age INT
    )`
    _, err := db.Exec(query)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Table created")
}
```

3. **Insert Data:** Write a function to insert data into the `users` table.

```go
func insertUser(db *sql.DB, name string, age int) {
    query := `INSERT INTO users (name, age) VALUES ($1, $2)`
    _, err := db.Exec(query, name, age)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("User inserted")
}
```

4. **Query Data:** Write a function to query and print all users.

```go
func queryUsers(db *sql.DB) {
    rows, err := db.Query("SELECT id, name, age FROM users")
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var name string
        var age int
        err := rows.Scan(&id, &name, &age)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Printf("ID: %d, Name: %s, Age: %d\n", id, name, age)
    }
}
```

OUTPUT:

```
amiraordiyeva@MacBook-Air-Amira golang % go run ex1.go
Table created
User inserted
User inserted
ID: 1, Name: Amira, Age: 19
ID: 2, Name: Kamila, Age: 20
```

Exercise 2: Working with PostgreSQL using `GORM`

Objective: Use GORM to perform similar operations as above, but with an ORM approach.

1. **Setup GORM:** Install GORM and the PostgreSQL driver:

```
amiraordiyeva@MacBook-Air-Amira golang % go get -u gorm.io/gorm
go get -u gorm.io/driver/postgres

go: downloading gorm.io/gorm v1.25.12
go: downloading github.com/jinzhu/now v1.1.5
go: downloading github.com/jinzhu/inflection v1.0.0
go: downloading golang.org/x/text v0.14.0
go: downloading golang.org/x/text v0.18.0
go: added github.com/jinzhu/inflection v1.0.0
go: added github.com/jinzhu/now v1.1.5
go: added golang.org/x/text v0.18.0
go: added gorm.io/gorm v1.25.12
go: downloading gorm.io/driver/postgres v1.5.9
go: downloading github.com/jackc/pgx/v5 v5.5.5
go: downloading github.com/jackc/pgx v3.6.2+incompatible
go: downloading github.com/jackc/pgx/v5 v5.7.1
go: downloading github.com/jackc/pgpassfile v1.0.0
go: downloading github.com/jackc/pgservicefile v0.0.0-20221227161230-091c0ba34f0a
go: downloading golang.org/x/crypto v0.17.0
go: downloading github.com/jackc/pgservicefile v0.0.0-20240606120523-5a60cdf6a761
go: downloading golang.org/x/crypto v0.27.0
go: downloading github.com/jackc/puddle/v2 v2.2.1
go: downloading github.com/jackc/puddle/v2 v2.2.2
go: downloading github.com/jackc/puddle v1.3.0
go: added github.com/jackc/pgpassfile v1.0.0
go: added github.com/jackc/pgservicefile v0.0.0-20240606120523-5a60cdf6a761
go: added github.com/jackc/pgx/v5 v5.7.1
go: added github.com/jackc/puddle/v2 v2.2.2
go: added golang.org/x/crypto v0.27.0
go: added gorm.io/driver/postgres v1.5.9
amiraordiyeva@MacBook-Air-Amira golang %
```

2. **Create a Model:** Define the `User` model that maps to the `users` table.

```go
package main
import(
    "gorm.io/gorm"

)
type User struct {
    ID   uint `gorm:"primaryKey"`
    Name string
    Age  int
}
```

3. **Auto Migrate:** Use GORM's `AutoMigrate` to create the `users` table based on the `User` struct.

```go
package main
import(
    "gorm.io/driver/postgres"
    "gorm.io/gorm"
    "log"

)
type User struct {
    ID   uint `gorm:"primaryKey"`
    Name string
    Age  int
}

func main() {
    connStr := "user=amiraordiyeva dbname=golang sslmode=disable"

    db, err := gorm.Open(postgres.Open(connStr), &gorm.Config{})
    if err != nil {
        log.Fatal("Failed to connect to the database:", err)
    }
    err = db.AutoMigrate(&User{})
    if err != nil {
        log.Fatal("Failed to migrate the database:", err)
    }

    log.Println("Database migrated")
}
```

4. **Insert Data:** Use GORM to insert users into the database.

```go
func createUsers(db *gorm.DB) {
    user1 := User{Name: "Amira", Age: 19}
    user2 := User{Name: "Kamila", Age: 20}
    db.Create(&user1)
    db.Create(&user2)

    log.Println("Users inserted")
}
```

5. **Query Data:** Use GORM to retrieve users from the database.

```go
func queryUsers(db *gorm.DB) {
    var users []User
    db.Find(&users)

    for _, user := range users {
        log.Printf("ID: %d, Name: %s, Age: %d\n", user.ID, user.Name
    }
}
```

OUTPUT:

```
2024/10/02 16:06:23 ID: 4, Name: Kamila, Age: 20
amiraordiyeva@MacBook-Air-Amira golang % go run ex2.go
2024/10/02 16:18:26 Users inserted
2024/10/02 16:18:26 ID: 1, Name: Amira, Age: 19
2024/10/02 16:18:26 ID: 2, Name: Kamila, Age: 20
2024/10/02 16:18:26 ID: 3, Name: Amira, Age: 19
2024/10/02 16:18:26 ID: 4, Name: Kamila, Age: 20
2024/10/02 16:18:26 ID: 5, Name: Amira, Age: 19
2024/10/02 16:18:26 ID: 6, Name: Kamila, Age: 20
```

Exercise 3: Rest API (make for both direct queries to database and gorm)

Create a REST API with routes for `GET`, `POST`, `PUT`, and `DELETE`.

Get Users (GET /users): A handler to fetch all users from the `users` table.

```go
func getUsers(w http.ResponseWriter, r *http.Request) {
    ageFilter := r.URL.Query().Get("age")
    sort := r.URL.Query().Get("sort")

    query := "SELECT id, name, age FROM users"
    if ageFilter != "" {
        query += " WHERE age = " + ageFilter
    }
    if sort == "name" {
        query += " ORDER BY name"
    }

    rows, err := db.Query(query)
    if err != nil {
        http.Error(w, "Ошибка выполнения запроса к базе данных", http.StatusInternalServerError)
        return
    }
    defer rows.Close()

    var users []User
    for rows.Next() {
        var user User
        if err := rows.Scan(&user.ID, &user.Name, &user.Age); err != nil {
            log.Fatal("Ошибка сканирования строки:", err)
        }
        users = append(users, user)
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(users)
}
```

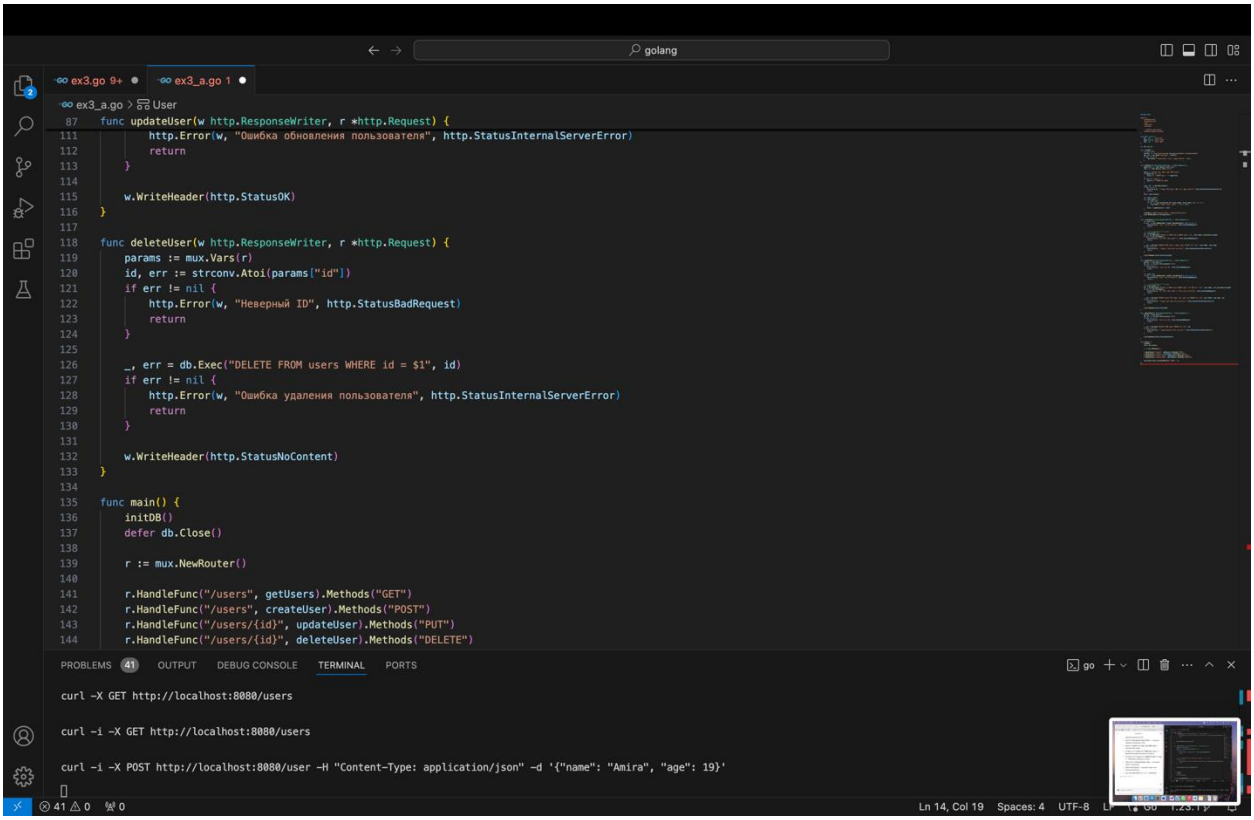
Create User (POST /user): A handler to insert a new user into the `users` table.

```go
func createUser(w http.ResponseWriter, r *http.Request) {
    var user User
    if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
        http.Error(w, "Некорректные данные", http.StatusBadRequest)
        return
    }

    // Проверка уникальности имени
    var existingID int
    err := db.QueryRow("SELECT id FROM users WHERE name = $1", user.Name).Scan(&existingID)
    if err != sql.ErrNoRows {
        http.Error(w, "Имя уже используется", http.StatusBadRequest)
        return
    }

    _, err = db.Exec("INSERT INTO users (name, age) VALUES ($1, $2)", user.Name, user.Age)
    if err != nil {
        http.Error(w, "Ошибка вставки пользователя", http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated)
}
```

Update User (PUT /user/{id}): A handler to update an existing user in the `users` table.

```go
func updateUser(w http.ResponseWriter, r *http.Request) {
    params := mux.Vars(r)
    id, err := strconv.Atoi(params["id"])
    if err != nil {
        http.Error(w, "Неверный ID", http.StatusBadRequest)
        return
    }

    var user User
    if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
        http.Error(w, "Некорректные данные", http.StatusBadRequest)
        return
    }

    // Проверка уникальности имени
    var existingID int
    err = db.QueryRow("SELECT id FROM users WHERE name = $1 AND id != $2", user.Name, id).Scan(&existingID)
    if err != sql.ErrNoRows {
        http.Error(w, "Имя уже используется другим пользователем", http.StatusBadRequest)
        return
    }

    _, err = db.Exec("UPDATE users SET name = $1, age = $2 WHERE id = $3", user.Name, user.Age, id)
    if err != nil {
        http.Error(w, "Ошибка обновления пользователя", http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusOK)
}
```

Delete User (DELETE /user/{id}): A handler to delete a user from the `users` table.



OUTPUT:


```
[{"ID":3,"Name":"Amira","Age":19},{"ID":4,"Name":"Kamila","Age":20},
{"ID":5,"Name":"Amira","Age":19},{"ID":6,"Name":"Kamila","Age":20},
{"ID":7,"Name":"Amira","Age":19},{"ID":8,"Name":"Kamila","Age":20},
{"ID":9,"Name":"Amira","Age":19},{"ID":10,"Name":"Kamila","Age":20},
{"ID":11,"Name":"Amira","Age":19},{"ID":12,"Name":"Kamila","Age":20},
{"ID":13,"Name":"Amira","Age":19},{"ID":14,"Name":"Kamila","Age":20},
{"ID":1,"Name":"Amira Updated","Age":20}]
```

Exercise 1: Advanced PostgreSQL Operations with `database/sql`

Objective: Connect to PostgreSQL, perform advanced operations, and handle transactions and error management.

1. **Setup PostgreSQL Connection:**
    - Create a Go program that connects to your PostgreSQL database using the `pq` driver.
    - Implement connection pooling with `sql.DB`.

```go
package main

import (
    "database/sql"
    "log"

    _ "github.com/lib/pq"
)

type User struct {
    ID   int
    Name string
    Age  int
}

func main() {
    connStr := "user=amiraordiyeva dbname=golang sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal("Ошибка подключения к базе данных:", err)
    }
    defer db.Close()

    if err := db.Ping(); err != nil {
        log.Fatal("Ошибка проверки подключения к базе данных:", err)
    }

    log.Println("Подключение к базе данных успешно!")
```

2. **Create a Table with Constraints:**
    - Write a function to create a `users` table with the following constraints:
        - `id` as a primary key, auto-incremented.
        - `name` as a unique, non-null field.
        - `age` as a non-null integer field.

```go
func createTable(db *sql.DB) {
    query := `
    CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name VARCHAR(100) UNIQUE NOT NULL,
        age INT NOT NULL
    );`
    _, err := db.Exec(query)
    if err != nil {
        log.Fatal("Ошибка создания таблицы:", err)
    }
    log.Println("Таблица users создана.")
}
```

3. **Insert Data with Transactions:**
    - Write a function to insert multiple users into the `users` table within a transaction.
    - Implement error handling to roll back the transaction if any error occurs during insertion.

```go
func insertUsers(db *sql.DB, users []User) {
    tx, err := db.Begin()
    if err != nil {
        log.Fatal("Ошибка начала транзакции:", err)
    }

    for _, user := range users {
        _, err := tx.Exec("INSERT INTO users (name, age) VALUE
        if err != nil {
            tx.Rollback()
            log.Fatal("Ошибка вставки пользователя:", err)
        }
    }

    err = tx.Commit()
    if err != nil {
        log.Fatal("Ошибка подтверждения транзакции:", err)
    }
    log.Println("Пользователи вставлены.")
}
```

4. **Query Data with Filtering and Pagination:**
    - Write a function to query and print users with optional filters for `age` and pagination support.
    - Implement pagination to return a specific number of results per page.

```go
func queryUsers(db *sql.DB, ageFilter int, page int, pageSize int) {
    var query string
    var args []interface{}
    if ageFilter > 0 {
        query = "SELECT * FROM users WHERE age = $1 LIMIT $2 OFFSET $3"
        args = []interface{}{ageFilter, pageSize, (page - 1) * pageSize}
    } else {
        query = "SELECT * FROM users LIMIT $1 OFFSET $2"
        args = []interface{}{pageSize, (page - 1) * pageSize}
    }

    rows, err := db.Query(query, args...)
    if err != nil {
        log.Fatal("Ошибка выполнения запроса:", err)
    }
    defer rows.Close()

    for rows.Next() {
        var user User
        if err := rows.Scan(&user.ID, &user.Name, &user.Age); err != nil {
            log.Fatal("Ошибка сканирования строки:", err)
        }
        log.Printf("ID: %d, Name: %s, Age: %d\n", user.ID, user.Name, user.Age)
    }
}
```

5. **Update and Delete Data:**
    - Write functions to update a user's details and delete a user by their ID, including error handling.

```go
func updateUser(db *sql.DB, id int, name string, age int) {
    _, err := db.Exec("UPDATE users SET name = $1, age = $2 WHERE id = $3", name, age, id)
    if err != nil {
        log.Fatal("Ошибка обновления пользователя:", err)
    }
    log.Println("Пользователь обновлен.")
}

func deleteUser(db *sql.DB, id int) {
    _, err := db.Exec("DELETE FROM users WHERE id = $1", id)
    if err != nil {
        log.Fatal("Ошибка удаления пользователя:", err)
    }
    log.Println("Пользователь удален.")
}

```

OUTPUT:

```
amiraordiyeva@MacBook-Air-Amira golang % go run ex1_a.go
2024/10/02 17:58:55 Подключение к базе данных успешно!
2024/10/02 17:58:55 Таблица users создана.
2024/10/02 17:58:55 Пользователи вставлены.
2024/10/02 17:58:55 ID: 3, Name: Amira, Age: 19
2024/10/02 17:58:55 ID: 4, Name: Kamila, Age: 20
2024/10/02 17:58:55 Пользователь обновлен.
2024/10/02 17:58:55 Пользователь удален.
```

Exercise 2: Advanced GORM Operations

Objective: Utilize GORM for more advanced operations including transactions, associations, and validation.

1. **Setup GORM with PostgreSQL:**
   - Install GORM and the PostgreSQL driver.
   - Configure GORM with connection pooling.

```
amiraordiyeva@MacBook-Air-Amira golang % go get -u gorm.io/gorm
go get -u gorm.io/driver/postgres

go: downloading github.com/jackc/pgx v3.6.2+incompatible
go: downloading github.com/jackc/puddle v1.3.0
```

2. **Create a Model with Associations:**
   - Define a `User` model with fields and add an associated `Profile` model. For example:
     - `User` with fields: `ID`, `Name`, `Age`.
     - `Profile` with fields: `ID`, `UserID`, `Bio`, `ProfilePictureURL`.
   - Set up the one-to-one association between `User` and `Profile`.

```go
type User struct {
    ID   uint   `gorm:"primaryKey"`
    Name string `gorm:"not null"`
    Age  int    `gorm:"not null"`
    Profile Profile
}

type Profile struct {
    ID               uint   `gorm:"primaryKey"`
    UserID           uint   `gorm:"not null;unique"`
    Bio              string
    ProfilePictureURL string
}
```

3. **Auto Migrate with Constraints and Associations:**
   - Use GORM's AutoMigrate to create tables for `User` and `Profile` with appropriate constraints and associations.

```
err = db.AutoMigrate(&User{}, &Profile{})
if err != nil {
    log.Fatal("Ошибка миграции базы данных:", err)
}

log.Println("Таблицы User и Profile созданы.")
```

4. **Insert Data with Associations:**
   - Use GORM to insert a `User` and an associated `Profile` in a single transaction.

```
err = db.AutoMigrate(&User{}, &Profile{})
if err != nil {
    log.Fatal("Ошибка миграции базы данных:", err)
}

log.Println("Таблицы User и Profile созданы.")
```

5. **Query Data with Associations:**
   Use GORM to retrieve users along with their profiles. Implement eager loading to optimize queries
   Функция, которая вставляет пользователя вместе с профилем в одну транзакцию.

```
func createUserWithProfile(db *gorm.DB, user User, profile Profile) {
    err := db.Transaction(func(tx *gorm.DB) error {
        if err := tx.Create(&user).Error; err != nil {
            return err
        }
        profile.UserID = user.ID
        if err := tx.Create(&profile).Error; err != nil {
            return err
        }
        return nil
    })
    if err != nil {
        log.Fatal("Ошибка вставки пользователя и профиля:", err)
    }
    log.Println("Пользователь и профиль успешно вставлены.")
}
```

Использую eager loading, чтобы получать пользователей вместе с их профилями.

```
func queryUsersWithProfiles(db *gorm.DB) {
    var users []User
    db.Preload("Profile").Find(&users)

    for _, user := range users {
        log.Printf("ID: %d, Name: %s, Age: %d, Bio: %s\n", user.ID, user.Name, user.Age, user.Profile.Bio)
    }
}
```

6. **Update and Delete Data:**
    - Write functions to update a user's profile and delete a user with associated profile, ensuring referential integrity.

```
func updateUserProfile(db *gorm.DB, userID uint, newBio string) {
    db.Model(&Profile{}).Where("user_id = ?", userID).Update("bio", newBio)
    log.Println("Профиль обновлен.")
}

func deleteUserWithProfile(db *gorm.DB, userID uint) {
    db.Transaction(func(tx *gorm.DB) error {
        if err := tx.Where("user_id = ?", userID).Delete(&Profile{}).Error; err != nil {
            return err
        }
        if err := tx.Delete(&User{}, userID).Error; err != nil {
            return err
        }
        return nil
    })
    log.Println("Пользователь и профиль удалены.")
}
```

Exercise 3: REST API with Advanced Features

Objective: Create a REST API with both direct `database/sql` queries and GORM, including additional features like filtering and sorting.

1. **Create REST API Routes with Direct SQL Queries:**
    - **Get Users (GET /users):** Fetch all users with optional query parameters for filtering by `age` and sorting by `name`.

```go
func main() {
}

func getUsersSQL(c *gin.Context) {
    age := c.Query("age")
    sort := c.Query("sort")
    limit := c.DefaultQuery("limit", "10")
    offset := c.DefaultQuery("offset", "0")

    var users []User
    query := "SELECT id, name, age FROM users"

    if age != "" {
        query += " WHERE age = " + age
    }
    if sort != "" {
        query += " ORDER BY " + sort
    }
    query += " LIMIT " + limit + " OFFSET " + offset

    rows, err := sqlDB.Query(query)
    handleError(c, err)
    defer rows.Close()

    for rows.Next() {
        var user User
        err := rows.Scan(&user.ID, &user.Name, &user.Age)
        handleError(c, err)
        users = append(users, user)
    }

    c.JSON(http.StatusOK, users)
}
```

- **Create User (POST /users):** Insert a new user with validation to ensure `name` is unique.

```go
func createUserSQL(c *gin.Context) {
    var user User
    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    var existingUser User
    err := sqlDB.QueryRow("SELECT id, name FROM users WHERE name=$1", user.Name).Scan(&existingUser.ID, &existingUser.Name)
    if err == nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Name already exists"})
        return
    }

    _, err = sqlDB.Exec("INSERT INTO users (name, age) VALUES ($1, $2)", user.Name, user.Age)
    handleError(c, err)

    c.JSON(http.StatusCreated, user)
}
```

- **Update User (PUT /users/{id}):** Update an existing user by ID with validation for `name` uniqueness.

```go
func updateUserSQL(c *gin.Context) {
    id := c.Param("id")
    var user User
    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    var existingUser User
    err := sqlDB.QueryRow("SELECT id, name FROM users WHERE name=$1 AND id != $2", user.Name, id).Scan(&existingUser.ID, &existingUser.Name)
    if err == nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Name already exists"})
        return
    }

    _, err = sqlDB.Exec("UPDATE users SET name=$1, age=$2 WHERE id=$3", user.Name, user.Age, id)
    handleError(c, err)

    c.JSON(http.StatusOK, user)
}
```

- **Delete User (DELETE /users/{id}):** Delete a user by ID, ensuring the ID exists.

```go
func deleteUserSQL(c *gin.Context) {
    id := c.Param("id")

    _, err := sqlDB.Exec("DELETE FROM users WHERE id=$1", id)
    handleError(c, err)

    c.Status(http.StatusNoContent)
}
```

2. **Create REST API Routes with GORM:**
   - **Get Users (GET /users):** Use GORM to fetch all users with filtering and sorting options.

```go
func getUsersGORM(c *gin.Context) {
    var users []User
    age := c.Query("age")
    sort := c.Query("sort")
    limit := c.DefaultQuery("limit", "10")
    offset := c.DefaultQuery("offset", "0")

    query := db.Model(&User{})

    if age != "" {
        query = query.Where("age = ?", age)
    }
    if sort != "" {
        query = query.Order(sort)
    }

    query.Limit(limit).Offset(offset).Find(&users)
    c.JSON(http.StatusOK, users)
}
```

   - **Create User (POST /users):** Use GORM to insert a new user with validation.

```go
func createUserGORM(c *gin.Context) {
    var user User
    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    var existingUser User
    if db.Where("name = ?", user.Name).First(&existingUser).RowsAffected > 0 {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Name already exists"})
        return
    }

    if err := db.Create(&user).Error; err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    c.JSON(http.StatusCreated, user)
}
```

- **Update User (PUT /users/{id}):** Use GORM to update an existing user by ID.

```go
func updateUserGORM(c *gin.Context) {
    id := c.Param("id")
    var user User

    if err := db.First(&user, id).Error; err != nil {
        c.JSON(http.StatusNotFound, gin.H{"error": "User not found"})
        return
    }

    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    var existingUser User
    if db.Where("name = ?", user.Name).Not("id = ?", id).First(&existingUser).RowsAffected > 0 {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Name already exists"})
        return
    }

    db.Save(&user)
    c.JSON(http.StatusOK, user)
}
```

- **Delete User (DELETE /users/{id}):** Use GORM to delete a user by ID.

```go
func deleteUserGORM(c *gin.Context) {
    id := c.Param("id")
    var user User

    if err := db.First(&user, id).Error; err != nil {
        c.JSON(http.StatusNotFound, gin.H{"error": "User not found"})
        return
    }

    db.Delete(&user)
    c.Status(http.StatusNoContent)
}
```

3. **Add Pagination and Error Handling:**
    - Implement pagination for the `GET /users` route for both direct SQL and GORM approaches.

```go
func main() {
}

func getUsersSQL(c *gin.Context) {
    age := c.Query("age")
    sort := c.Query("sort")
    limit := c.DefaultQuery("limit", "10")
    offset := c.DefaultQuery("offset", "0")

    var users []User
    query := "SELECT id, name, age FROM users"

    if age != "" {
        query += " WHERE age = " + age
    }
    if sort != "" {
        query += " ORDER BY " + sort
    }
    query += " LIMIT " + limit + " OFFSET " + offset

    rows, err := sqlDB.Query(query)
    handleError(c, err)
    defer rows.Close()

    for rows.Next() {
        var user User
        err := rows.Scan(&user.ID, &user.Name, &user.Age)
        handleError(c, err)
        users = append(users, user)
    }

    c.JSON(http.StatusOK, users)
}
```

    - Add comprehensive error handling for all API endpoints, including validation errors and database errors.

```go
func handleError(c *gin.Context, err error) {
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }
}
```

4. **Testing and Documentation:**
    - Write unit tests for each API endpoint.
    - Document the API using Swagger or another API documentation tool.