Assignment 2, Go

Put all deliverables into github repository in your profile. Share link to google form 24 hours before defense. Defend by explaining deliverables and answering questions.
Deliverables: code
Google form:
https://docs.google.com/forms/d/e/1FAIpQLSe0GyNdOYIvM1tX_I_CtlPod5jBf-ACLGdHYZq1gVZbUeBzIg/viewform?usp=sf_link

Exercise 1: Connecting to PostgreSQL Directly with `database/sql`

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

   Here are the connection credentials

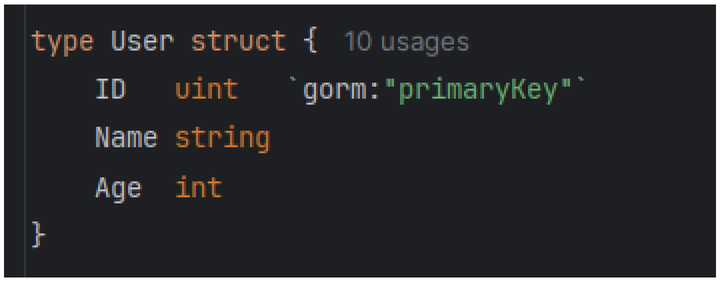
2. Create a Table: Write a function to create a simple table `users` with columns for `id`, `name`, and `age`.
3. **Insert Data:** Write a function to insert data into the `users` table.
4. **Query Data:** Write a function to query and print all users.

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

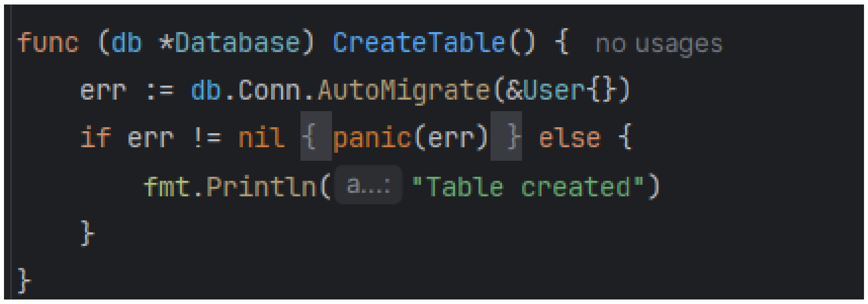
   Here is the struct for the table



```go
type User struct {  10 usages
    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.

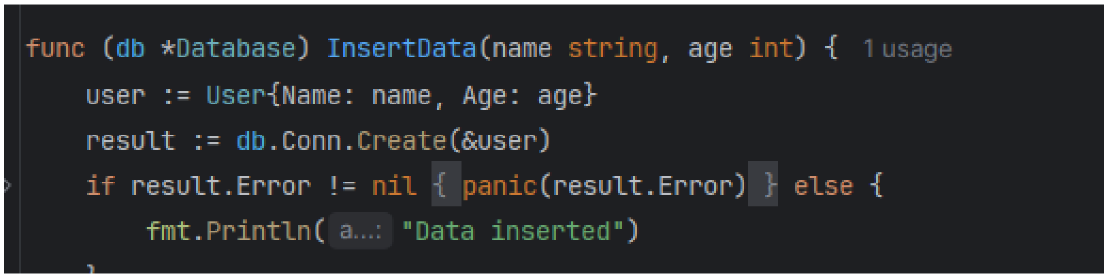
   AutoMigrate uses User struct to create the table



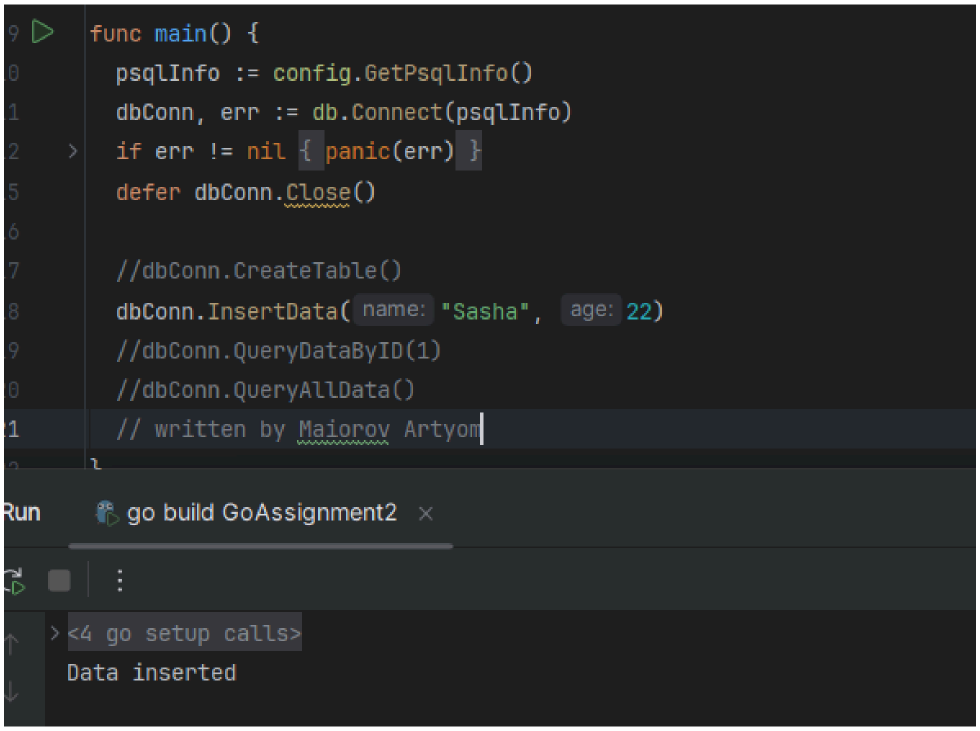
```go
func (db *Database) CreateTable() {  no usages
    err := db.Conn.AutoMigrate(&User{})
    if err != nil { panic(err) } else {
        fmt.Println( a...: "Table created")
    }
}
```

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

   InsertData takes 2 fields to create a User object and inserts it into the table



```go
func (db *Database) InsertData(name string, age int) {  1 usage
    user := User{Name: name, Age: age}
    result := db.Conn.Create(&user)
    if result.Error != nil { panic(result.Error) } else {
        fmt.Println( a...: "Data inserted")
```

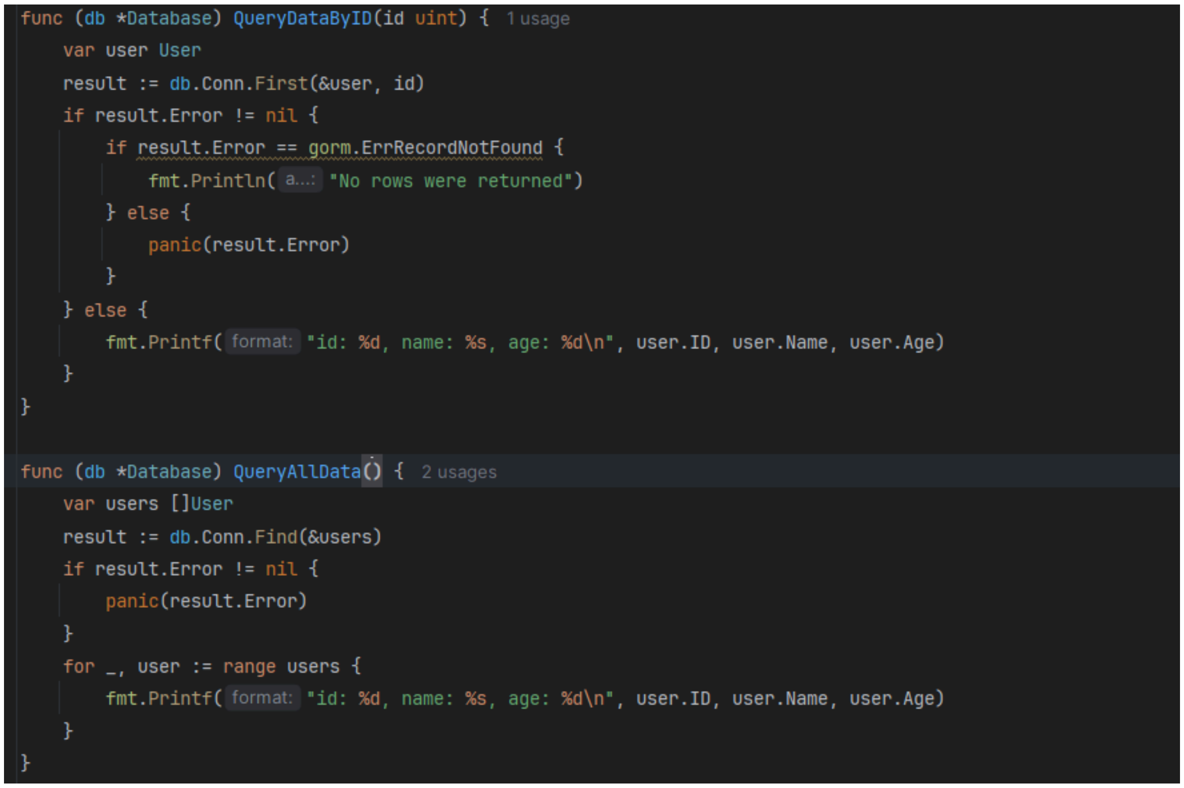


```go
func main() {
  psqlInfo := config.GetPsqlInfo()
  dbConn, err := db.Connect(psqlInfo)
  if err != nil { panic(err) }
  defer dbConn.Close()

  //dbConn.CreateTable()
  dbConn.InsertData( name: "Sasha",  age: 22)
  //dbConn.QueryDataByID(1)
  //dbConn.QueryAllData()
  // written by Maiorov Artyom
}
```

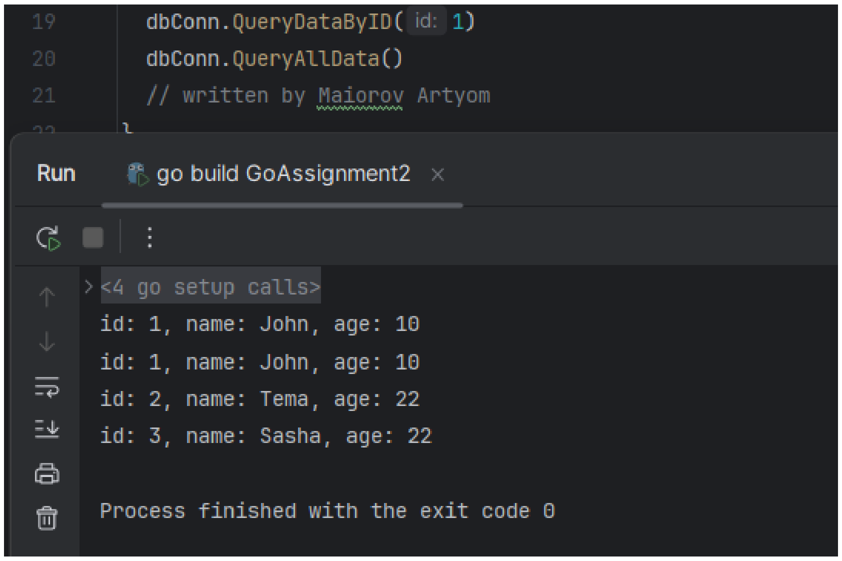
```
<4 go setup calls>
Data inserted
```

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

   These 2 methods retrieve users by id or retrieve all users and has error handling



```go
func (db *Database) QueryDataByID(id uint) {  1 usage
    var user User
    result := db.Conn.First(&user, id)
    if result.Error != nil {
        if result.Error == gorm.ErrRecordNotFound {
            fmt.Println( a...: "No rows were returned")
        } else {
            panic(result.Error)
        }
    } else {
        fmt.Printf( format: "id: %d, name: %s, age: %d\n", user.ID, user.Name, user.Age)
    }
}

func (db *Database) QueryAllData() {  2 usages
    var users []User
    result := db.Conn.Find(&users)
    if result.Error != nil {
        panic(result.Error)
    }
    for _, user := range users {
        fmt.Printf( format: "id: %d, name: %s, age: %d\n", user.ID, user.Name, user.Age)
    }
}
```



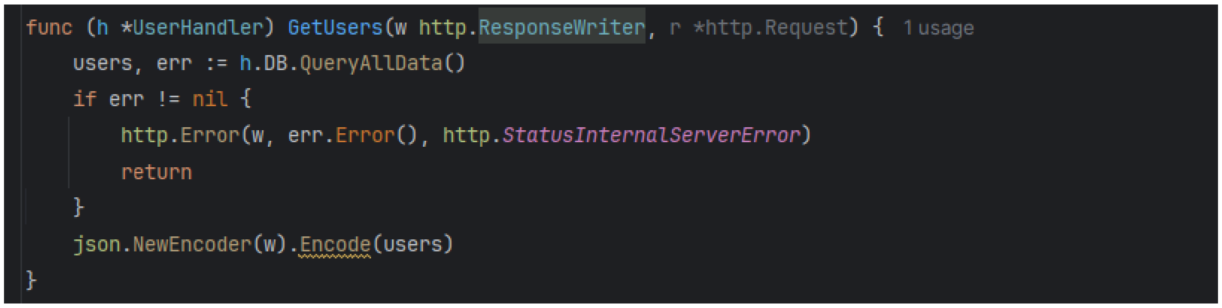
```
    dbConn.QueryDataByID( id: 1)
    dbConn.QueryAllData()
    // written by Maiorov Artyom
```

```
<4 go setup calls>
id: 1, name: John, age: 10
id: 1, name: John, age: 10
id: 2, name: Tema, age: 22
id: 3, name: Sasha, age: 22

Process finished with the exit code 0
```

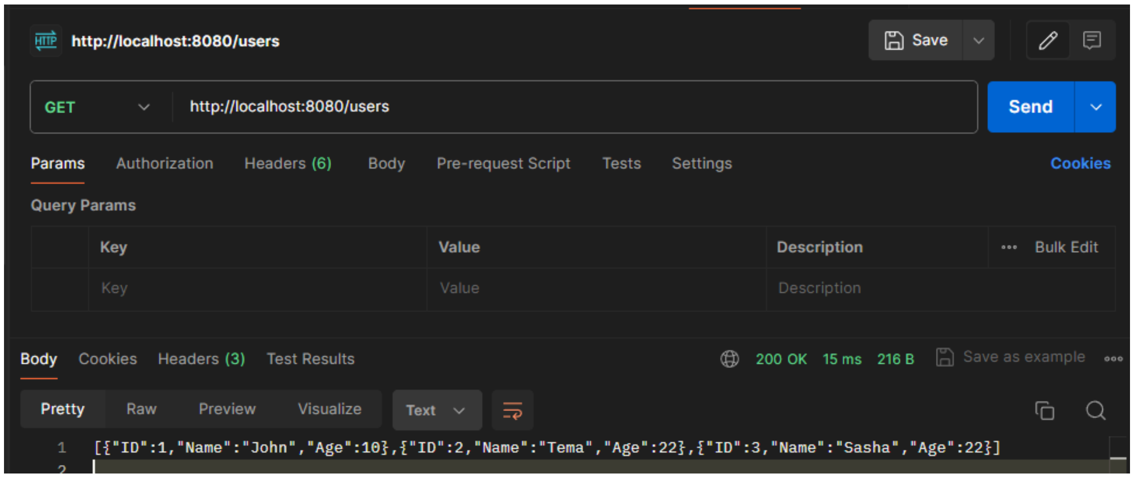
Exercise 3: Rest API (make for both direct queries to database and gorm)

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.

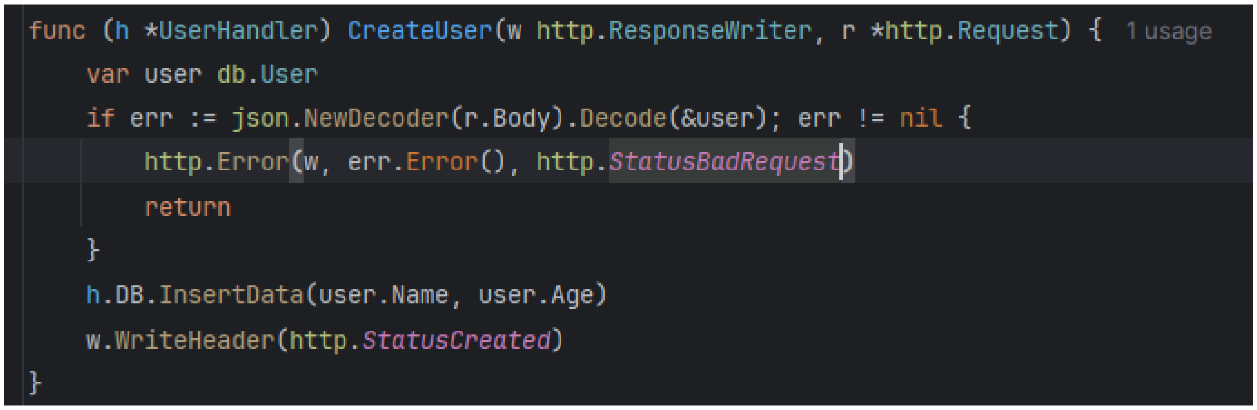


```go
func (h *UserHandler) GetUsers(w http.ResponseWriter, r *http.Request) {  1 usage
    users, err := h.DB.QueryAllData()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    json.NewEncoder(w).Encode(users)
}
```

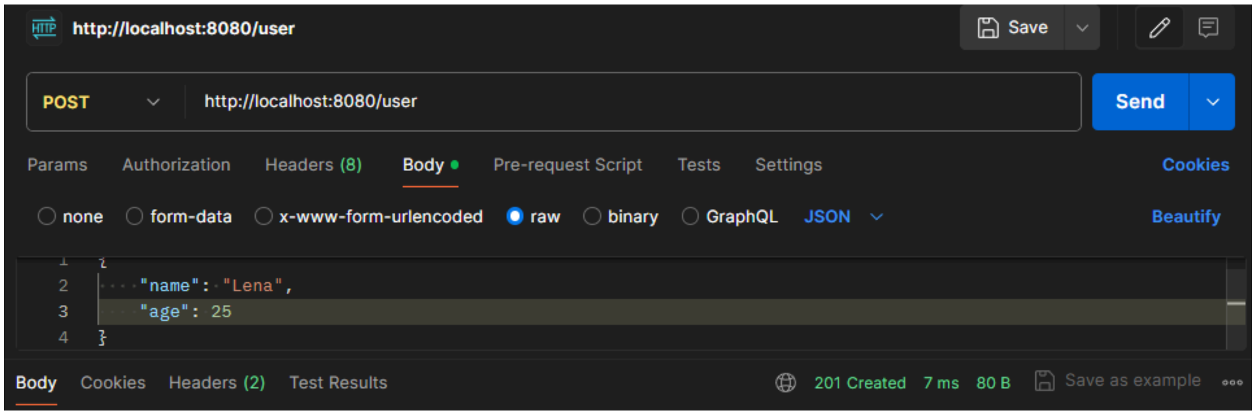


```
[{"ID":1,"Name":"John","Age":10},{"ID":2,"Name":"Tema","Age":22},{"ID":3,"Name":"Sasha","Age":22}]
```

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



```go
func (h *UserHandler) CreateUser(w http.ResponseWriter, r *http.Request) {  1 usage
    var user db.User
    if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    h.DB.InsertData(user.Name, user.Age)
    w.WriteHeader(http.StatusCreated)
}
```


http://localhost:8080/user

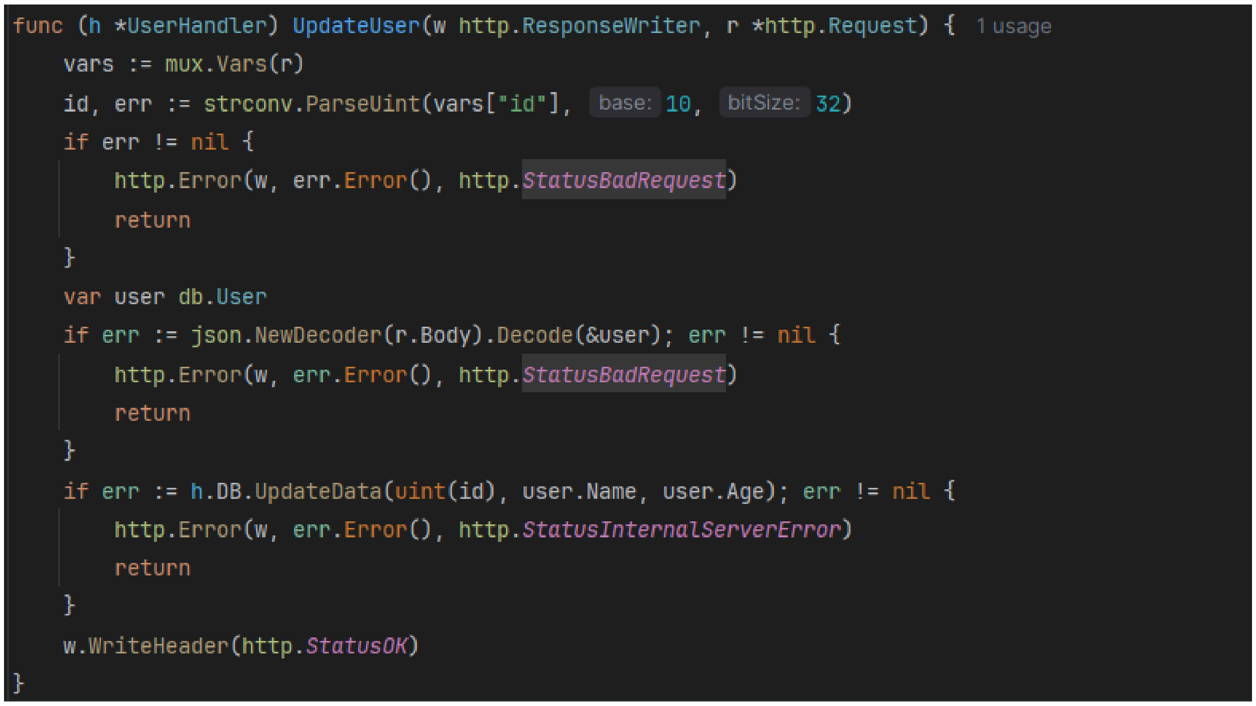
POST http://localhost:8080/user

Body: raw JSON

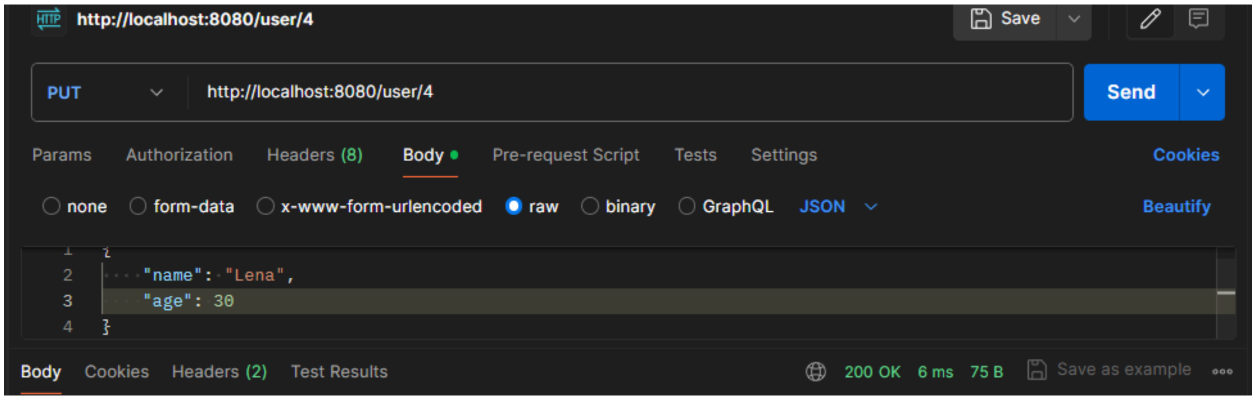
```
{
    "name": "Lena",
    "age": 25
}
```

201 Created 7 ms 80 B

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



```go
func (h *UserHandler) UpdateUser(w http.ResponseWriter, r *http.Request) {  1 usage
    vars := mux.Vars(r)
    id, err := strconv.ParseUint(vars["id"], base: 10, bitSize: 32)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    var user db.User
    if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    if err := h.DB.UpdateData(uint(id), user.Name, user.Age); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    w.WriteHeader(http.StatusOK)
}
```


http://localhost:8080/user/4

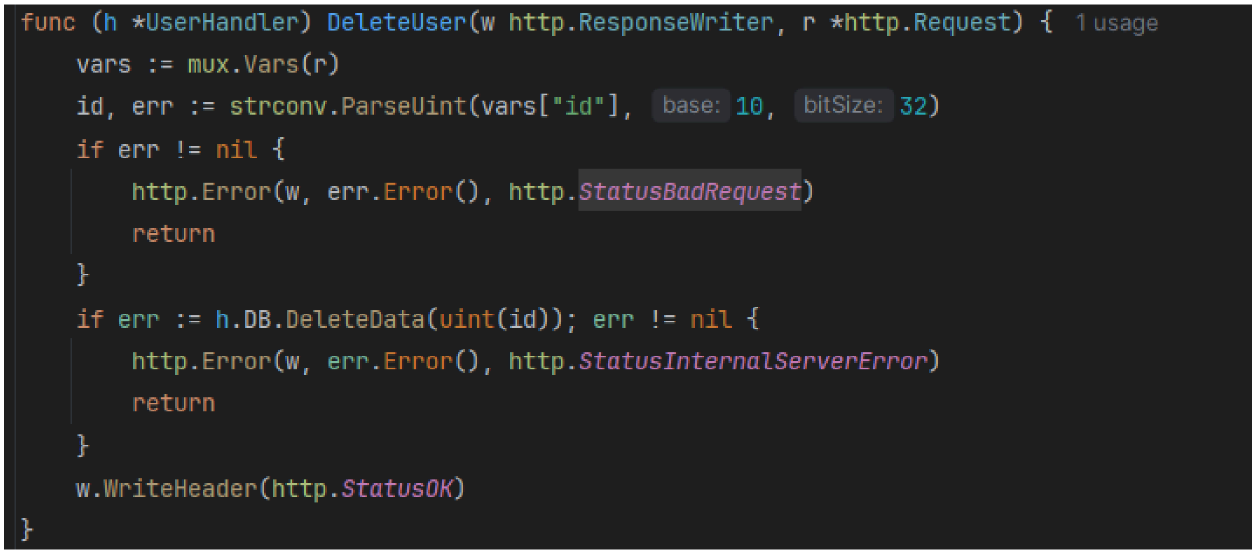
PUT http://localhost:8080/user/4

Body: raw JSON

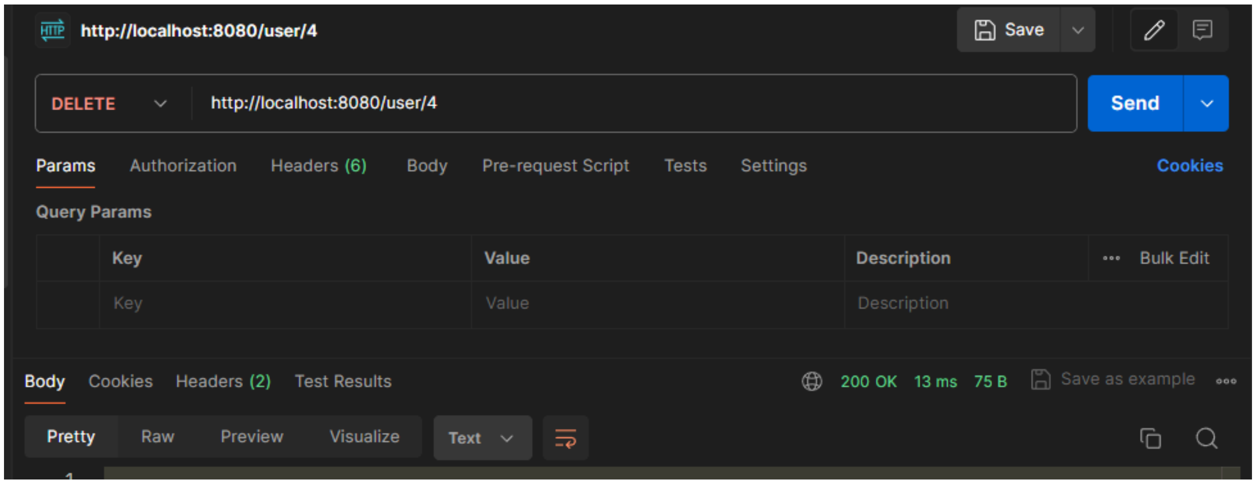
```
{
    "name": "Lena",
    "age": 30
}
```

200 OK 6 ms 75 B

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



```go
func (h *UserHandler) DeleteUser(w http.ResponseWriter, r *http.Request) {  1 usage
    vars := mux.Vars(r)
    id, err := strconv.ParseUint(vars["id"],  base: 10,  bitSize: 32)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    if err := h.DB.DeleteData(uint(id)); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    w.WriteHeader(http.StatusOK)
}
```



http://localhost:8080/user/4

DELETE http://localhost:8080/user/4

Params | Authorization | Headers (6) | Body | Pre-request Script | Tests | Settings | Cookies

Query Params

Key	Value	Description
Key	Value	Description

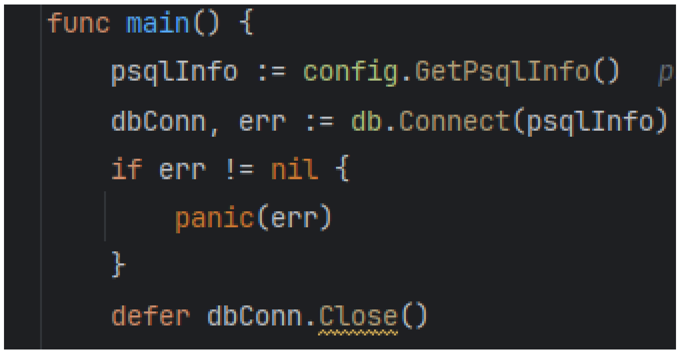
Body | Cookies | Headers (2) | Test Results — 200 OK 13 ms 75 B

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

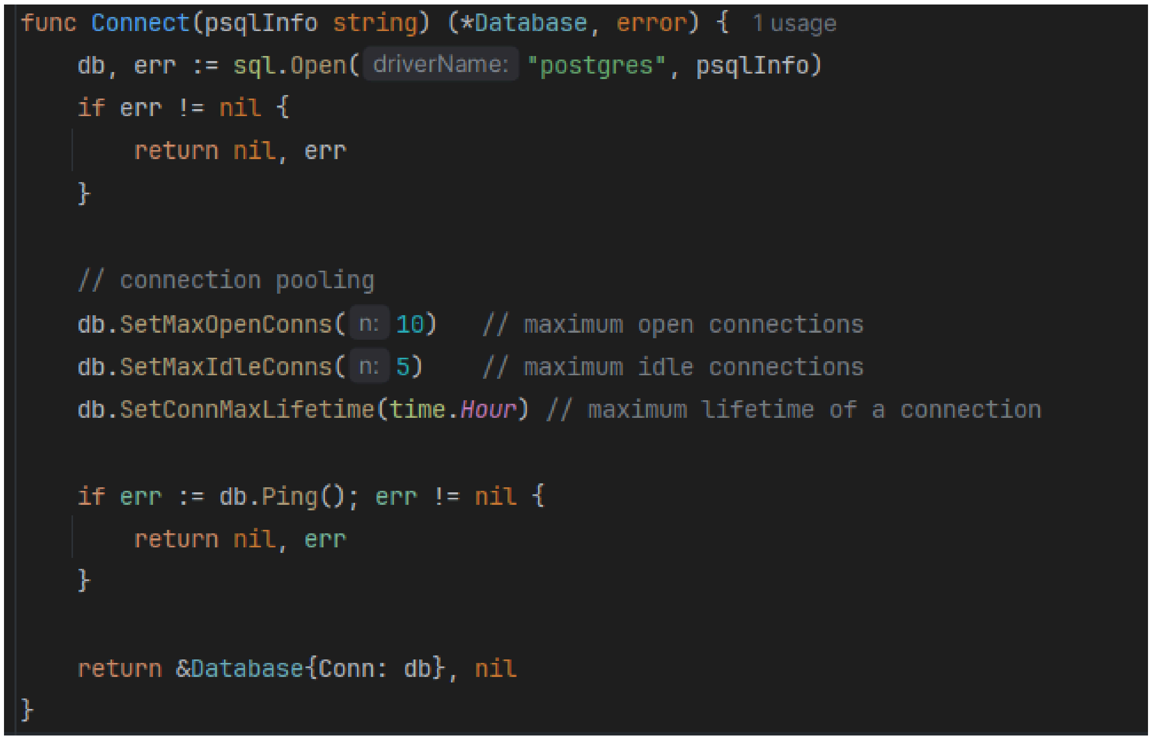
Objective: Connect to PostgreSQL, perform advanced operations, and handle transactions and error management.

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

    Creates a connection to a PSQL database and specifies connection pooling



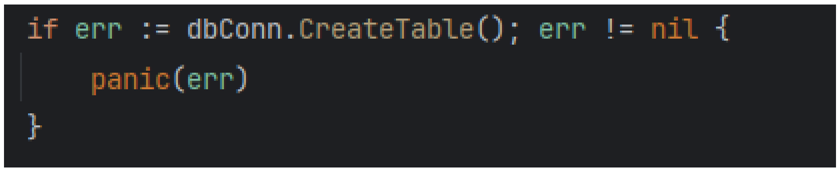
```go
func main() {
    psqlInfo := config.GetPsqlInfo()   p
    dbConn, err := db.Connect(psqlInfo)
    if err != nil {
        panic(err)
    }
    defer dbConn.Close()
```



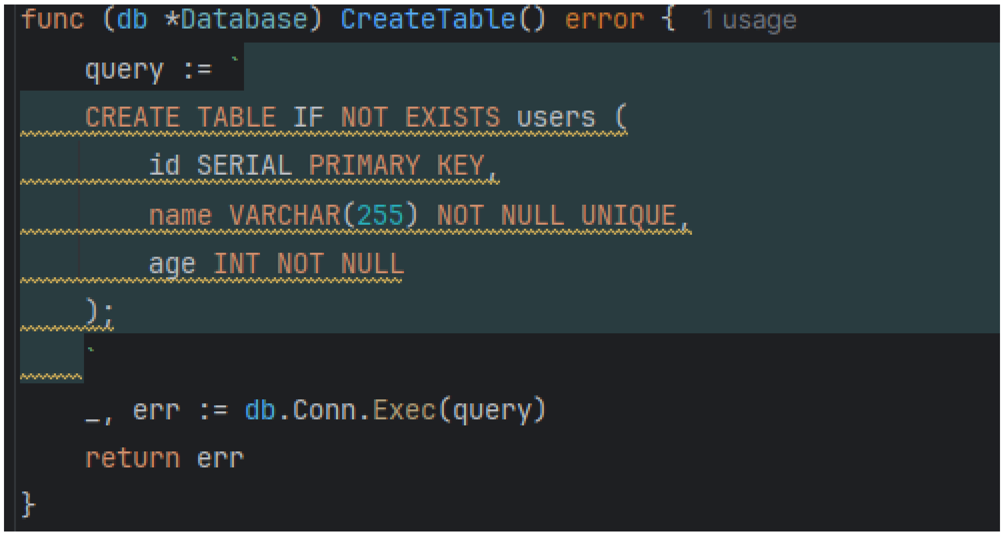
```go
func Connect(psqlInfo string) (*Database, error) {  1 usage
    db, err := sql.Open( driverName: "postgres", psqlInfo)
    if err != nil {
        return nil, err
    }

    // connection pooling
    db.SetMaxOpenConns( n: 10)    // maximum open connections
    db.SetMaxIdleConns( n: 5)     // maximum idle connections
    db.SetConnMaxLifetime(time.Hour) // maximum lifetime of a connection

    if err := db.Ping(); err != nil {
        return nil, err
    }

    return &Database{Conn: db}, nil
}
```

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.

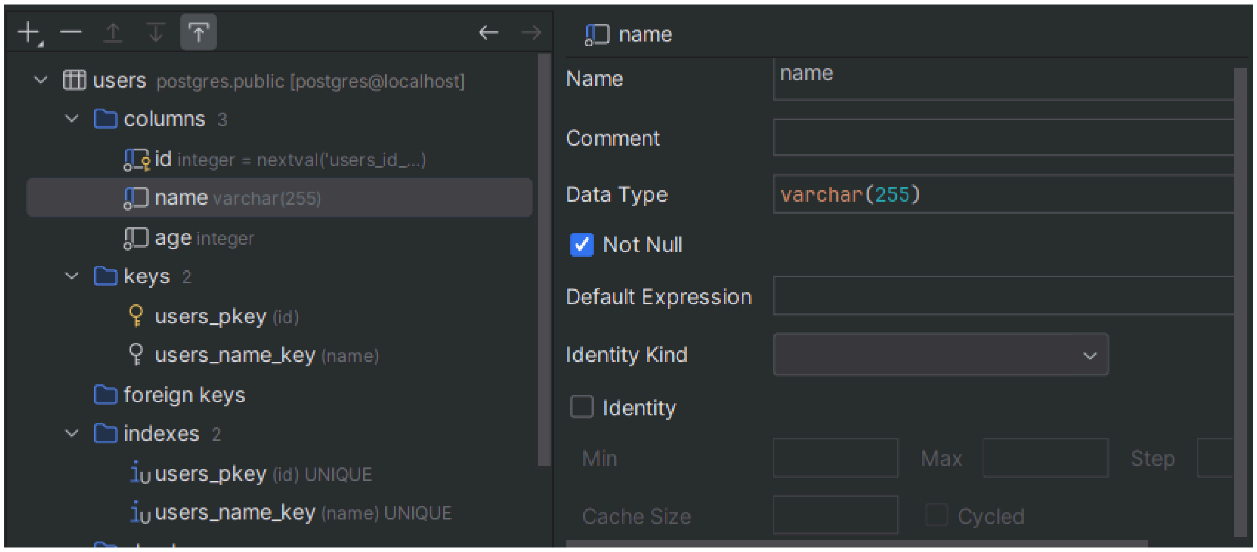
Creates a table users with constraints



```
if err := dbConn.CreateTable(); err != nil {
    panic(err)
}
```



```
func (db *Database) CreateTable() error {  1 usage
    query := `
    CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name VARCHAR(255) NOT NULL UNIQUE,
        age INT NOT NULL
    );
    `
    _, err := db.Conn.Exec(query)
    return err
}
```



```
users postgres.public [postgres@localhost]
    columns 3
        id integer = nextval('users_id_...)
        name varchar(255)
        age integer
    keys 2
        users_pkey (id)
        users_name_key (name)
    foreign keys
    indexes 2
        users_pkey (id) UNIQUE
        users_name_key (name) UNIQUE
```

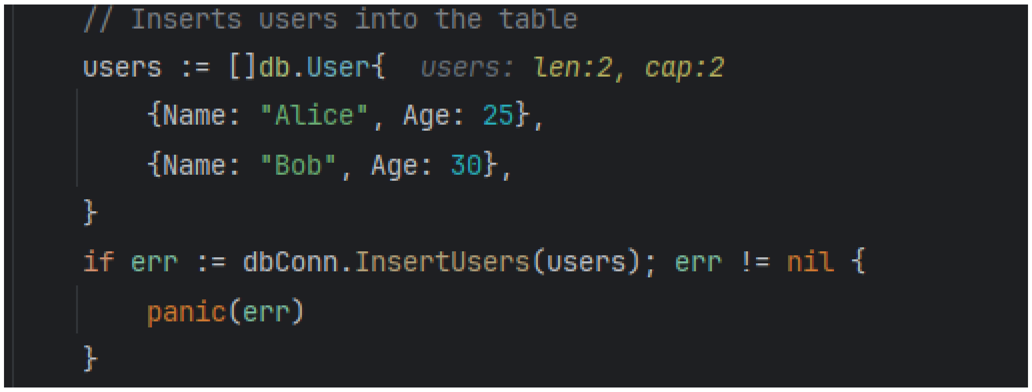
name

Name	name
Comment	
Data Type	varchar(255)
✓ Not Null	
Default Expression	
Identity Kind	
☐ Identity	
Min	Max	Step
Cache Size	☐ Cycled

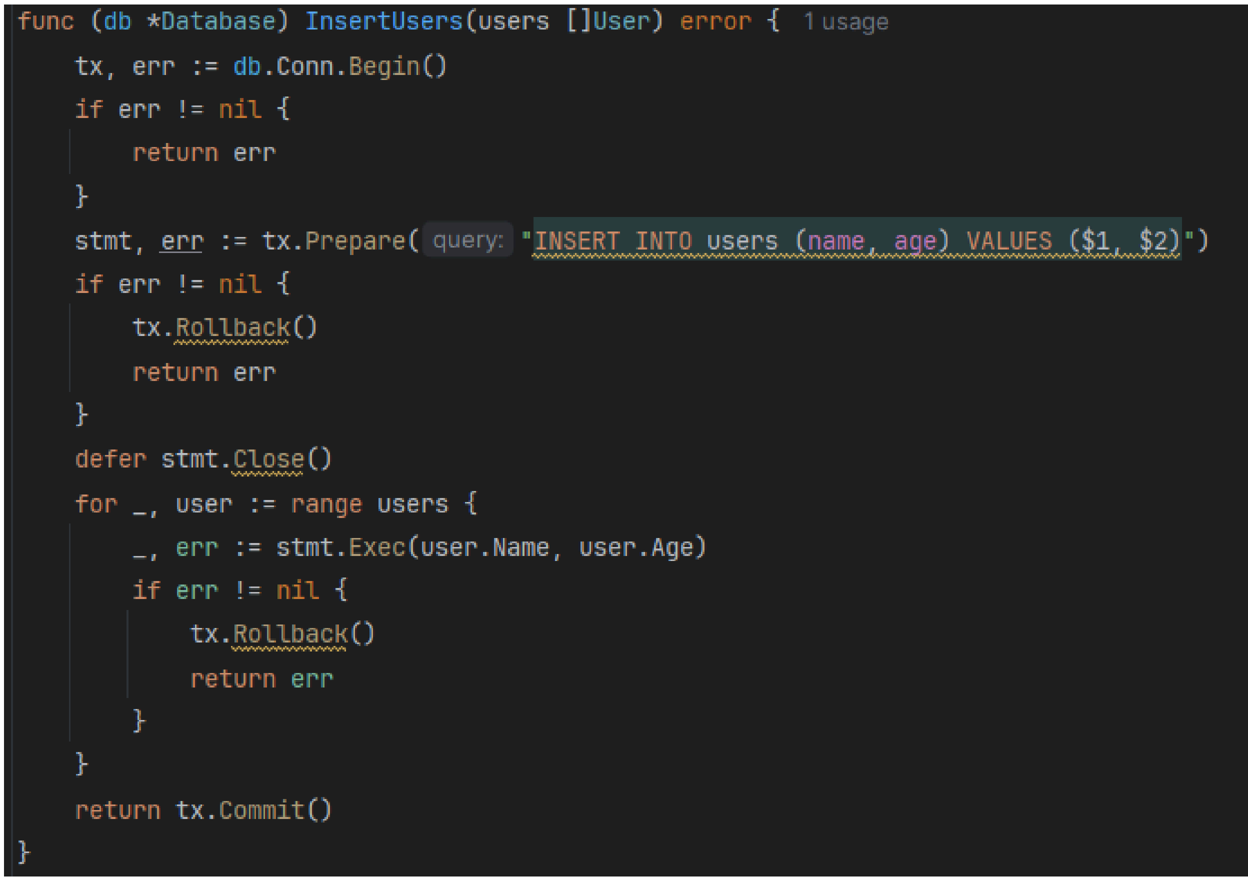
3. **Insert Data with Transactions:**

- 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.

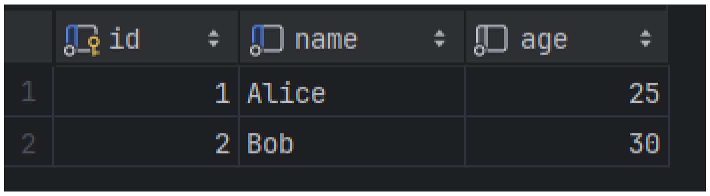
Inserts 2 users into the table



```go
// Inserts users into the table
users := []db.User{  users: len:2, cap:2
    {Name: "Alice", Age: 25},
    {Name: "Bob", Age: 30},
}
if err := dbConn.InsertUsers(users); err != nil {
    panic(err)
}
```

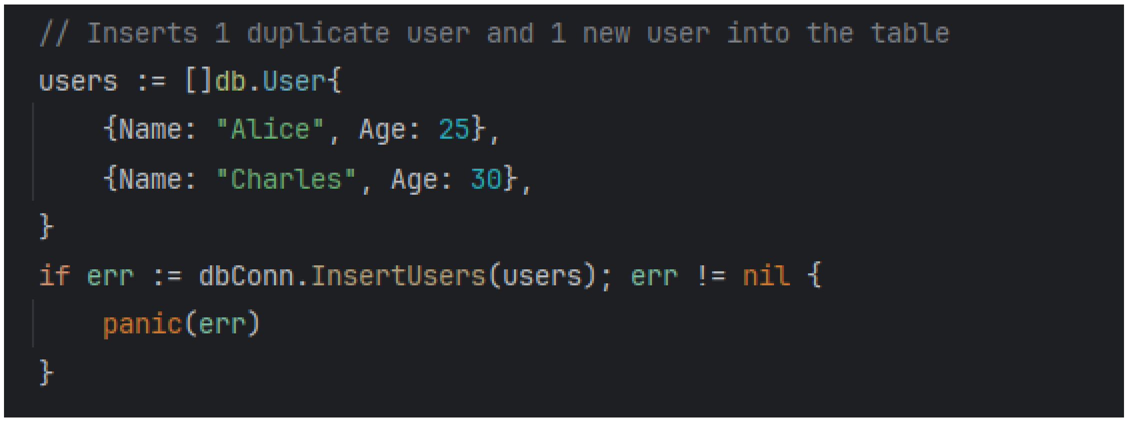


```go
func (db *Database) InsertUsers(users []User) error {  1 usage
    tx, err := db.Conn.Begin()
    if err != nil {
        return err
    }
    stmt, err := tx.Prepare( query: "INSERT INTO users (name, age) VALUES ($1, $2)")
    if err != nil {
        tx.Rollback()
        return err
    }
    defer stmt.Close()
    for _, user := range users {
        _, err := stmt.Exec(user.Name, user.Age)
        if err != nil {
            tx.Rollback()
            return err
        }
    }
    return tx.Commit()
}
```

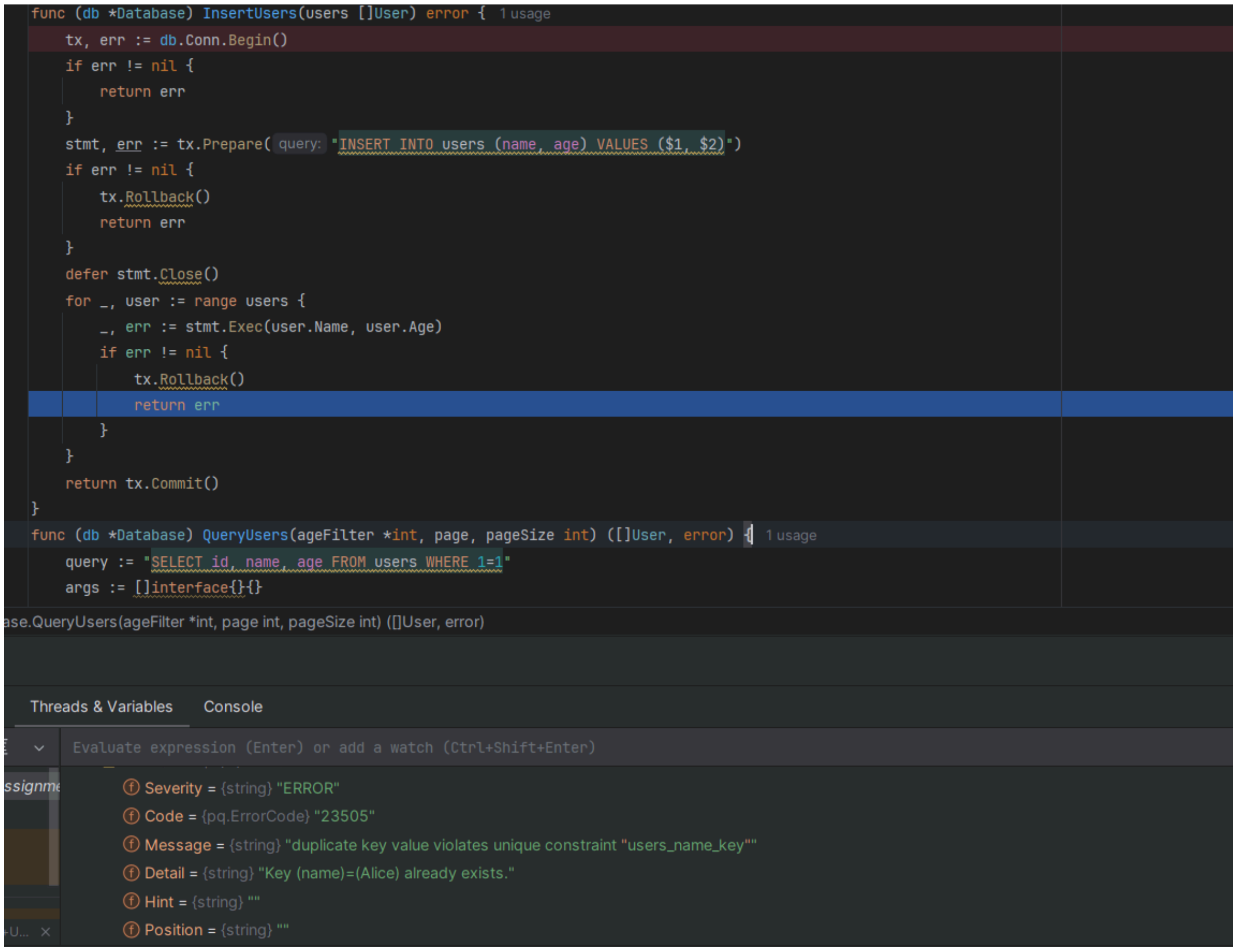


	id	name	age
1	1	Alice	25
2	2	Bob	30

If we try to violate the unique constraint by inserting a duplicate name, we get an error and the transaction is rolled back and the rest of the data is not inserted



```go
// Inserts 1 duplicate user and 1 new user into the table
users := []db.User{
    {Name: "Alice", Age: 25},
    {Name: "Charles", Age: 30},
}
if err := dbConn.InsertUsers(users); err != nil {
    panic(err)
}
```



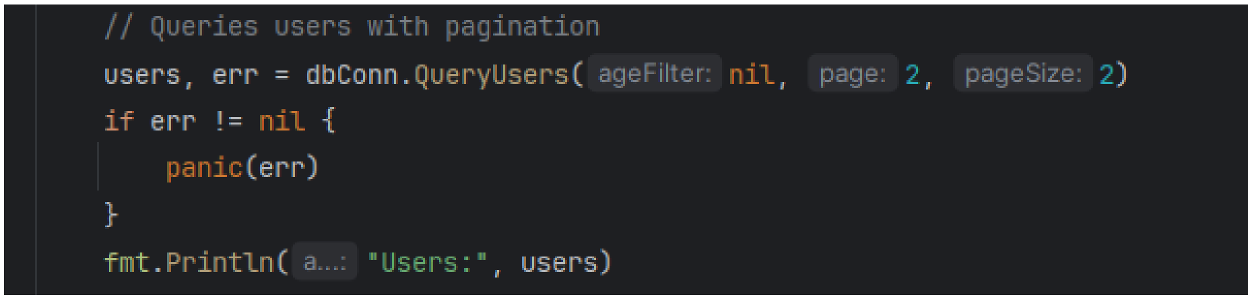
```
func (db *Database) InsertUsers(users []User) error {  1 usage
    tx, err := db.Conn.Begin()
    if err != nil {
        return err
    }
    stmt, err := tx.Prepare( query: "INSERT INTO users (name, age) VALUES ($1, $2)")
    if err != nil {
        tx.Rollback()
        return err
    }
    defer stmt.Close()
    for _, user := range users {
        _, err := stmt.Exec(user.Name, user.Age)
        if err != nil {
            tx.Rollback()
            return err
        }
    }
    return tx.Commit()
}
func (db *Database) QueryUsers(ageFilter *int, page, pageSize int) ([]User, error) {  1 usage
    query := "SELECT id, name, age FROM users WHERE 1=1"
    args := []interface{}{}
ase.QueryUsers(ageFilter *int, page int, pageSize int) ([]User, error)
```

Threads & Variables
Console
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
ssignme

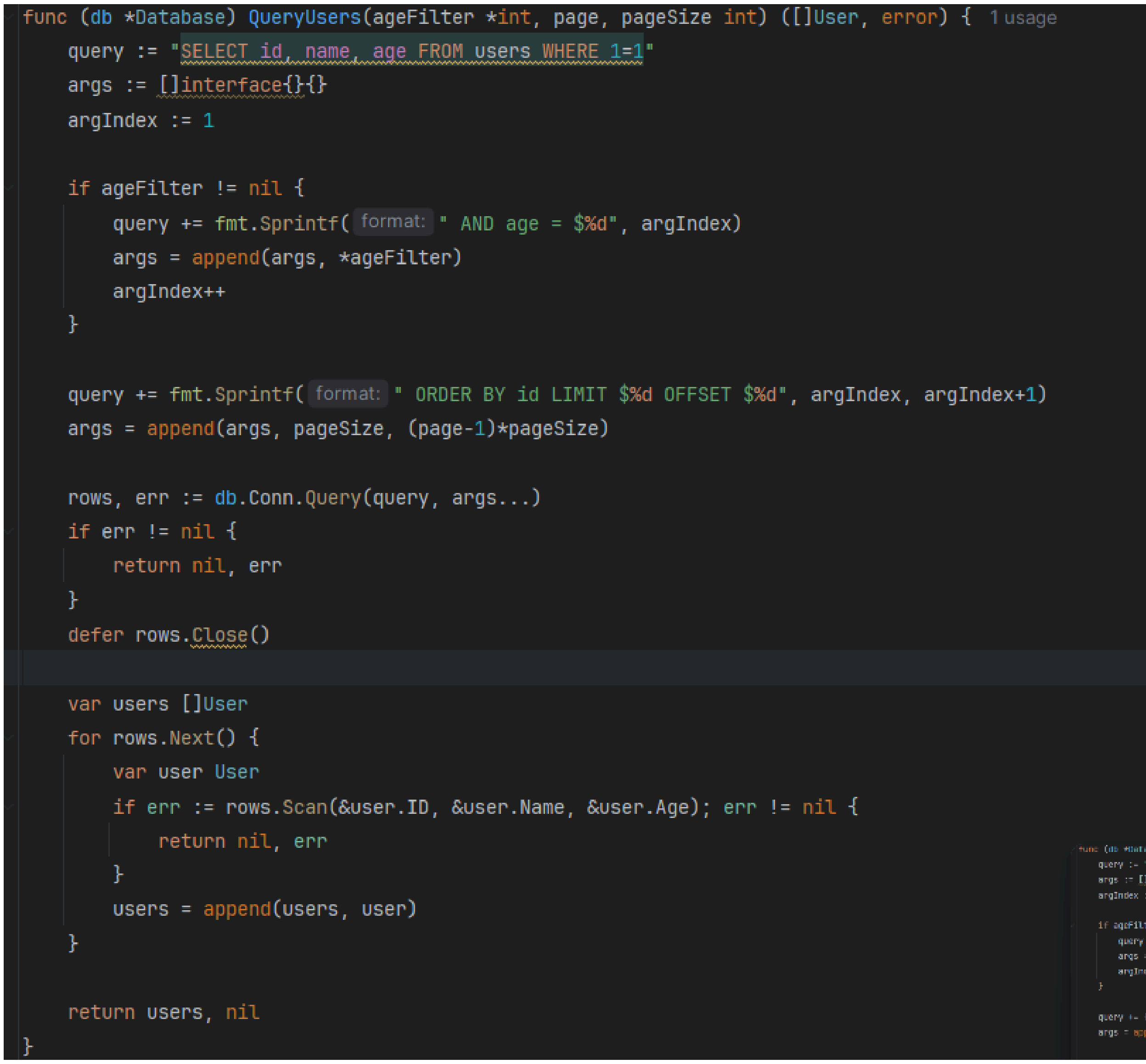
```
Severity = {string} "ERROR"
Code = {pq.ErrorCode} "23505"
Message = {string} "duplicate key value violates unique constraint "users_name_key""
Detail = {string} "Key (name)=(Alice) already exists."
Hint = {string} ""
Position = {string} ""
```

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.

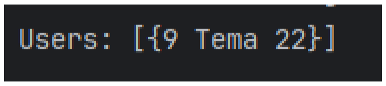
    The base query is changed dynamically based on ageFilter parameter and pagination is implemented with offset and limit.



```
// Queries users with pagination
users, err = dbConn.QueryUsers( ageFilter: nil,  page: 2,  pageSize: 2)
if err != nil {
    panic(err)
}
fmt.Println( a...: "Users:", users)
```



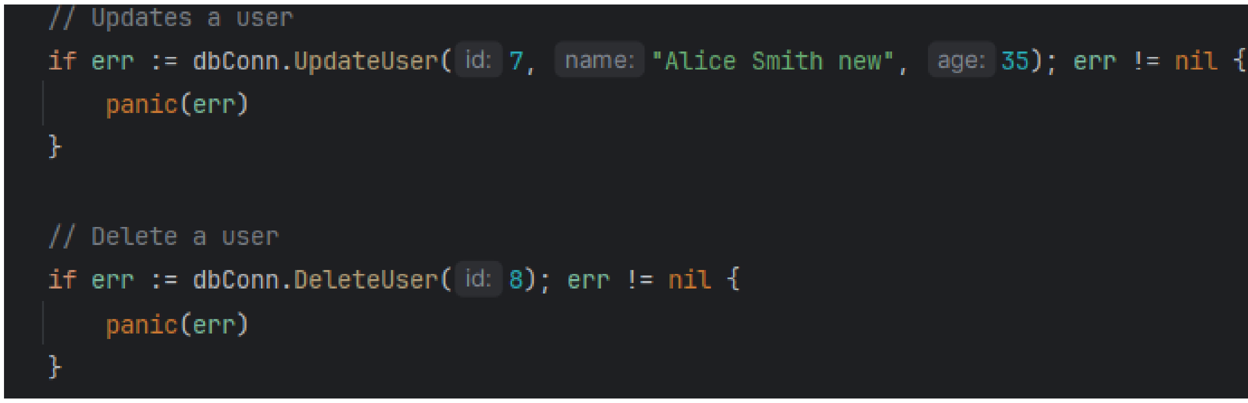
```
func (db *Database) QueryUsers(ageFilter *int, page, pageSize int) ([]User, error) {  1 usage
    query := "SELECT id, name, age FROM users WHERE 1=1"
    args := []interface{}{}
    argIndex := 1

    if ageFilter != nil {
        query += fmt.Sprintf( format: " AND age = $%d", argIndex)
        args = append(args, *ageFilter)
        argIndex++
    }

    query += fmt.Sprintf( format: " ORDER BY id LIMIT $%d OFFSET $%d", argIndex, argIndex+1)
    args = append(args, pageSize, (page-1)*pageSize)

    rows, err := db.Conn.Query(query, args...)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

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

    return users, nil
}
```



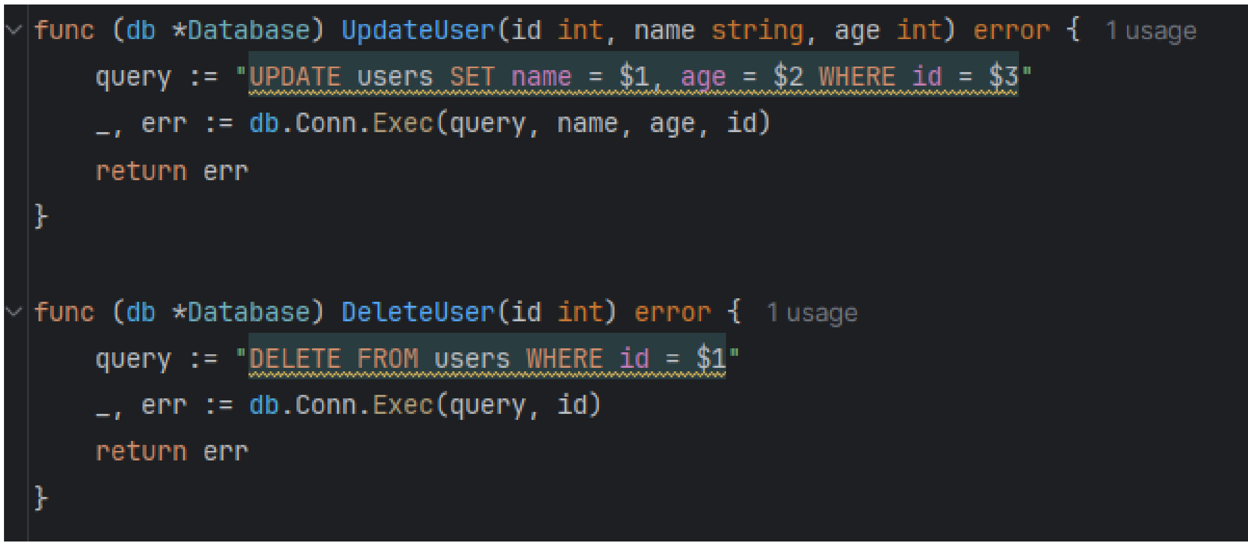
```
Users: [{9 Tema 22}]
```

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

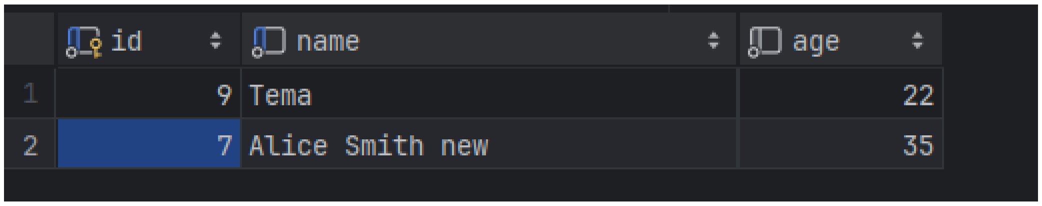
      Updates a user based on ID with specified values and Delete is also based on ID



```
// Updates a user
if err := dbConn.UpdateUser( id: 7, name: "Alice Smith new", age: 35); err != nil {
    panic(err)
}

// Delete a user
if err := dbConn.DeleteUser( id: 8); err != nil {
    panic(err)
}
```



```
func (db *Database) UpdateUser(id int, name string, age int) error {  1 usage
    query := "UPDATE users SET name = $1, age = $2 WHERE id = $3"
    _, err := db.Conn.Exec(query, name, age, id)
    return err
}

func (db *Database) DeleteUser(id int) error {  1 usage
    query := "DELETE FROM users WHERE id = $1"
    _, err := db.Conn.Exec(query, id)
    return err
}
```



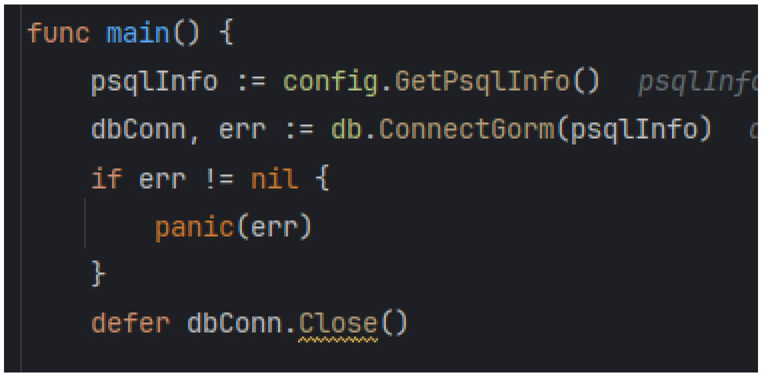
	id	name	age
1	9	Tema	22
2	7	Alice Smith new	35

Exercise 2: Advanced GORM Operations

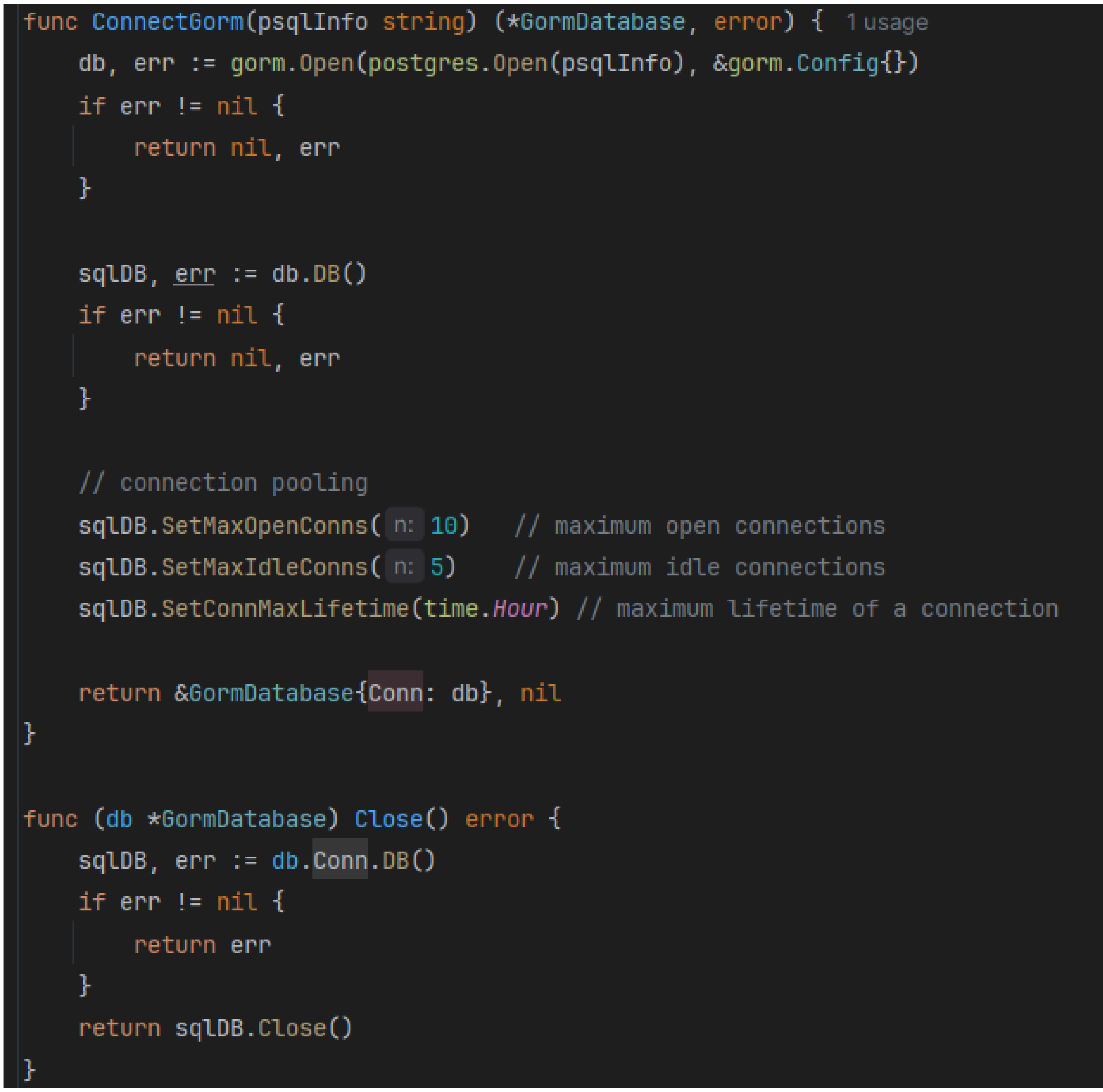
Objective: Utilize GORM for more advanced operations including transactions, associations, and validation.

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

    This code creates a connection to a PSQL database and sets up the connection pooling configurations



```
func main() {
    psqlInfo := config.GetPsqlInfo()   psqlInf
    dbConn, err := db.ConnectGorm(psqlInfo)   d
    if err != nil {
        panic(err)
    }
    defer dbConn.Close()
```

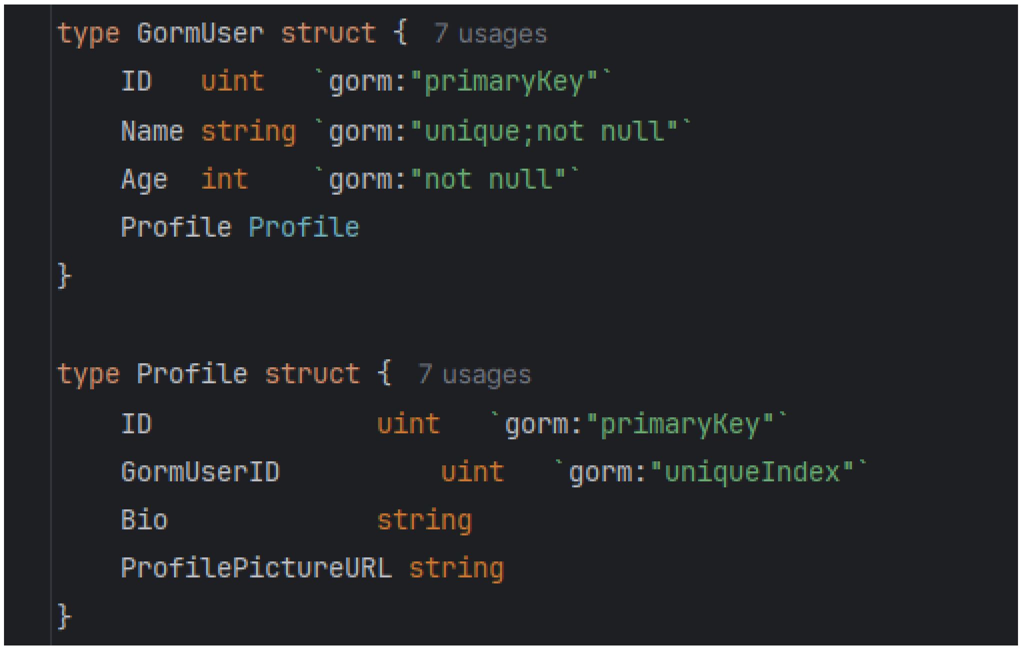


```
func ConnectGorm(psqlInfo string) (*GormDatabase, error) {  1 usage
    db, err := gorm.Open(postgres.Open(psqlInfo), &gorm.Config{})
    if err != nil {
        return nil, err
    }

    sqlDB, err := db.DB()
    if err != nil {
        return nil, err
    }

    // connection pooling
    sqlDB.SetMaxOpenConns( n: 10)   // maximum open connections
    sqlDB.SetMaxIdleConns( n: 5)    // maximum idle connections
    sqlDB.SetConnMaxLifetime(time.Hour) // maximum lifetime of a connection

    return &GormDatabase{Conn: db}, nil
}

func (db *GormDatabase) Close() error {
    sqlDB, err := db.Conn.DB()
    if err != nil {
        return err
    }
    return sqlDB.Close()
}
```

2. **Create a Model with Associations:**

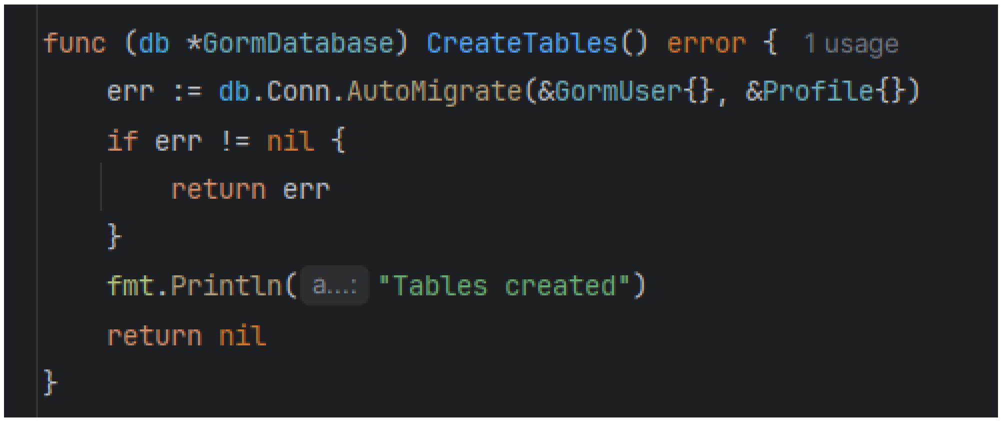
- Define a `User` model with fields and add an associated `Profile` model. For example:
  - `User` with fields: `ID`, `Name`, `Age`.
  - `Profile` with fields: `ID`, `UserID`, `Bio`, `ProfilePictureURL`.
- Set up the one-to-one association between `User` and `Profile`.

  The one-to-one link is in the UserID



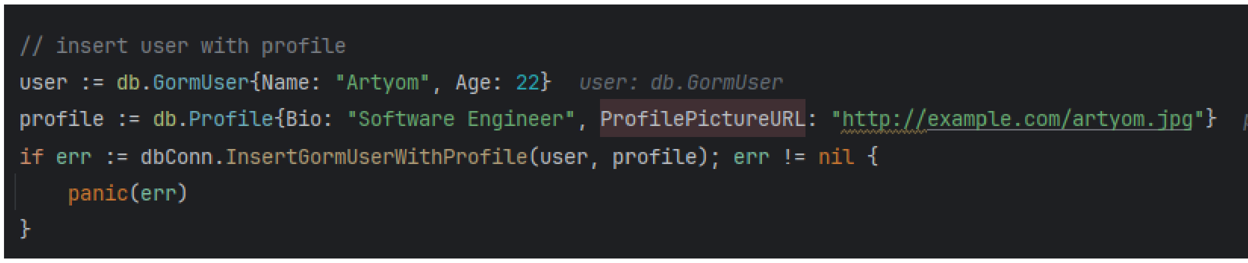
```go
type GormUser struct {  7 usages
    ID   uint   `gorm:"primaryKey"`
    Name string `gorm:"unique;not null"`
    Age  int    `gorm:"not null"`
    Profile Profile
}

type Profile struct {  7 usages
    ID                uint   `gorm:"primaryKey"`
    GormUserID           uint   `gorm:"uniqueIndex"`
    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.

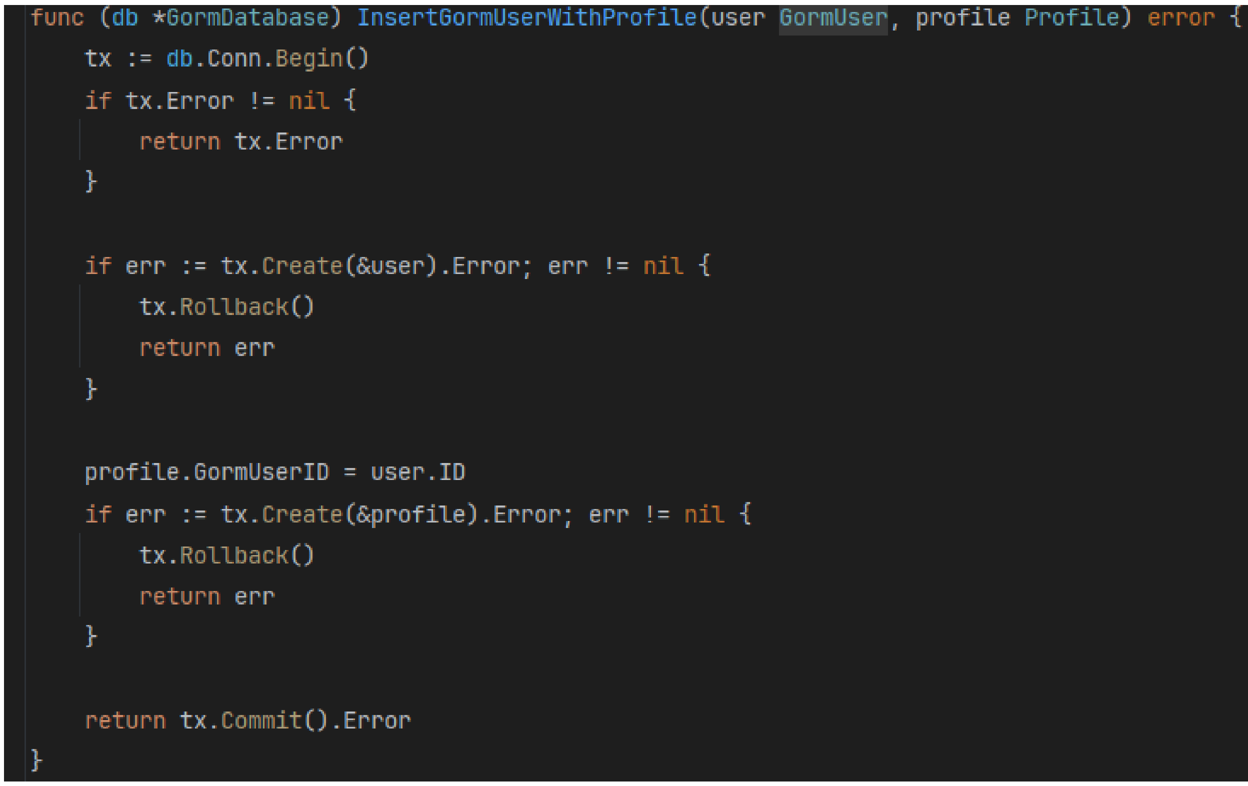


```go
func (db *GormDatabase) CreateTables() error {  1 usage
    err := db.Conn.AutoMigrate(&GormUser{}, &Profile{})
    if err != nil {
        return err
    }
    fmt.Println( a...: "Tables created")
    return nil
}
```

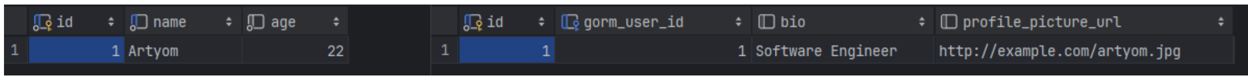
4. **Insert Data with Associations:**
   - Use GORM to insert a `User` and an associated `Profile` in a single transaction.



```go
// insert user with profile
user := db.GormUser{Name: "Artyom", Age: 22}  user: db.GormUser
profile := db.Profile{Bio: "Software Engineer", ProfilePictureURL: "http://example.com/artyom.jpg"}
if err := dbConn.InsertGormUserWithProfile(user, profile); err != nil {
    panic(err)
}
```



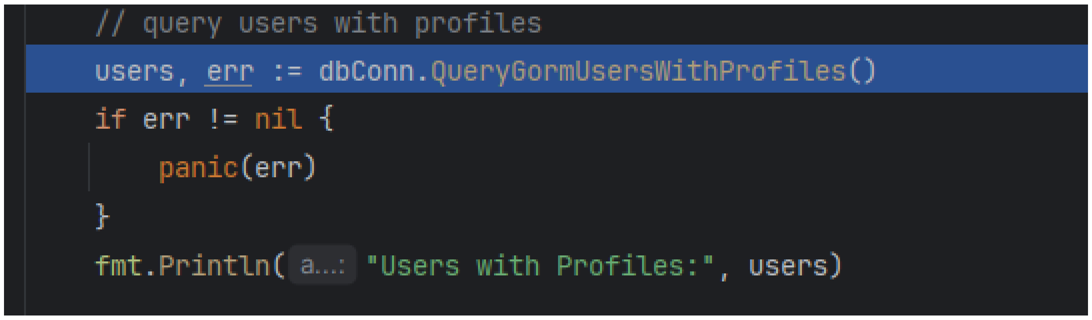
```go
func (db *GormDatabase) InsertGormUserWithProfile(user GormUser, profile Profile) error {
    tx := db.Conn.Begin()
    if tx.Error != nil {
        return tx.Error
    }

    if err := tx.Create(&user).Error; err != nil {
        tx.Rollback()
        return err
    }

    profile.GormUserID = user.ID
    if err := tx.Create(&profile).Error; err != nil {
        tx.Rollback()
        return err
    }

    return tx.Commit().Error
}
```



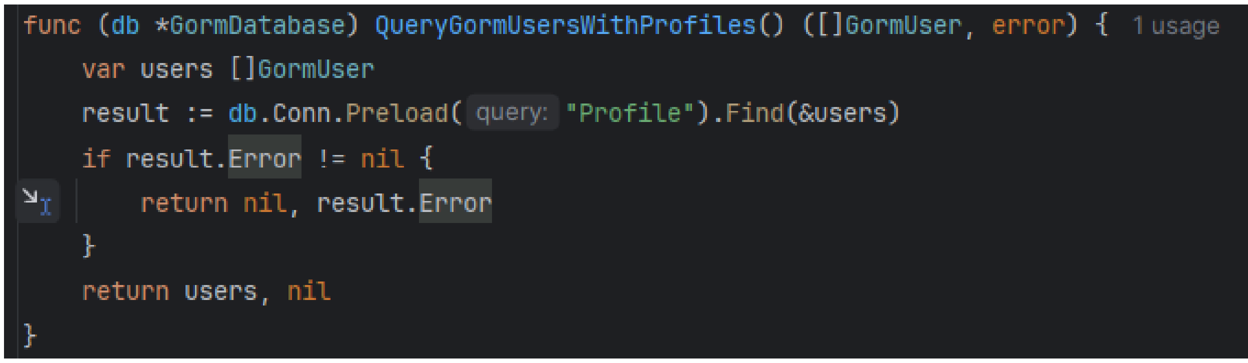
	id	name	age
1	1	Artyom	22

	id	gorm_user_id	bio	profile_picture_url
1	1	1	Software Engineer	http://example.com/artyom.jpg

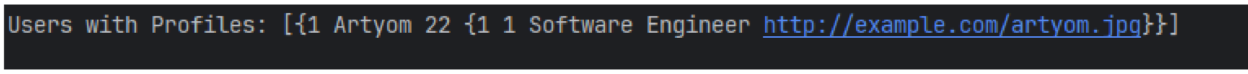
5. **Query Data with Associations:**
   - Use GORM to retrieve users along with their profiles. Implement eager loading to optimize queries.



```go
// query users with profiles
users, err := dbConn.QueryGormUsersWithProfiles()
if err != nil {
    panic(err)
}
fmt.Println( a...: "Users with Profiles:", users)
```

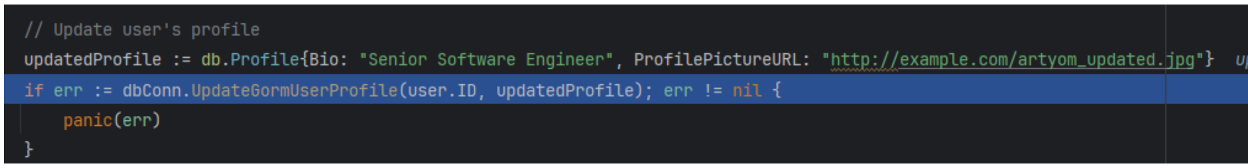


```go
func (db *GormDatabase) QueryGormUsersWithProfiles() ([]GormUser, error) {  1 usage
    var users []GormUser
    result := db.Conn.Preload( query: "Profile").Find(&users)
    if result.Error != nil {
        return nil, result.Error
    }
    return users, nil
}
```

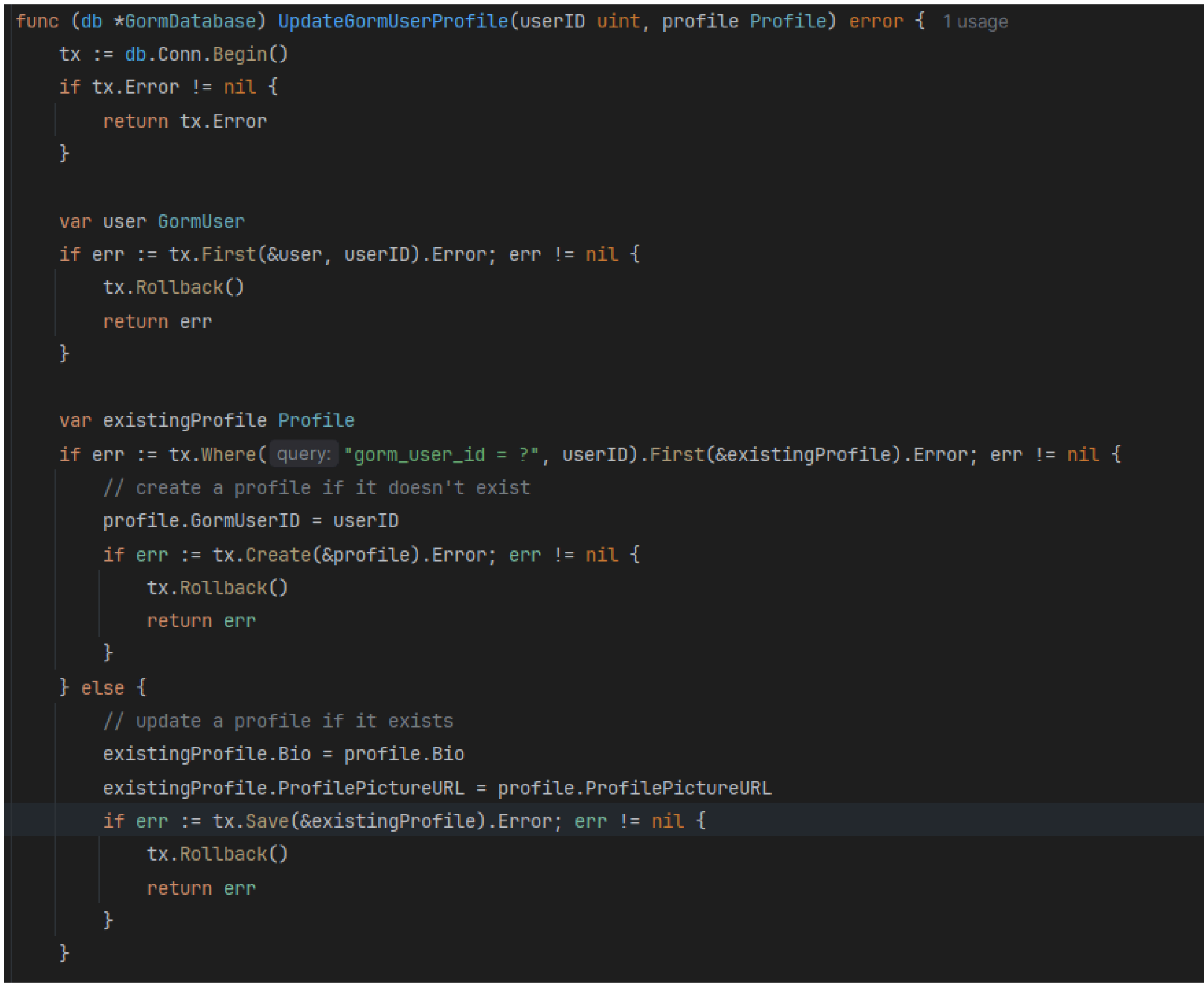


```
Users with Profiles: [{1 Artyom 22 {1 1 Software Engineer http://example.com/artyom.jpg}}]
```

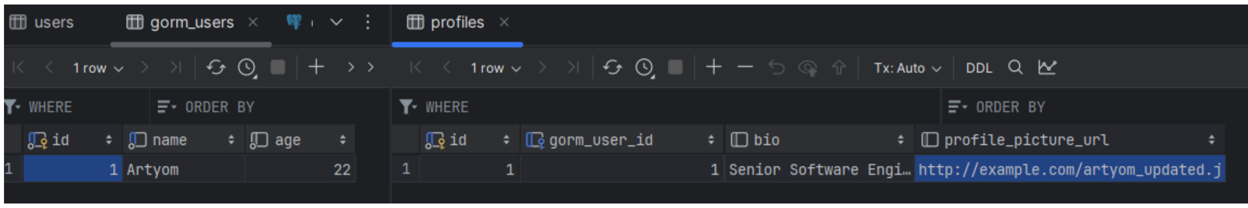
6. **Update and Delete Data:**
   - Write functions to update a user's profile and delete a user with associated profile, ensuring referential integrity.



```go
// Update user's profile
updatedProfile := db.Profile{Bio: "Senior Software Engineer", ProfilePictureURL: "http://example.com/artyom_updated.jpg"}
if err := dbConn.UpdateGormUserProfile(user.ID, updatedProfile); err != nil {
    panic(err)
}
```

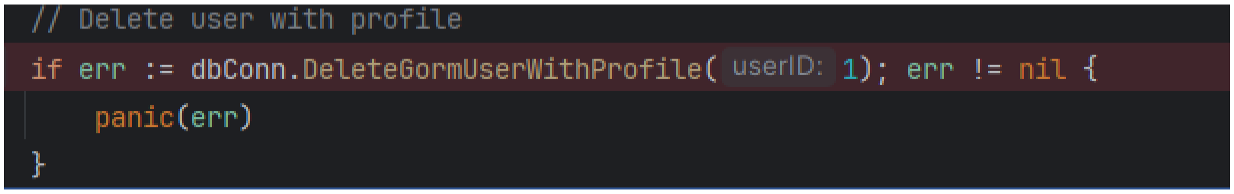


```go
func (db *GormDatabase) UpdateGormUserProfile(userID uint, profile Profile) error {  1 usage
    tx := db.Conn.Begin()
    if tx.Error != nil {
        return tx.Error
    }

    var user GormUser
    if err := tx.First(&user, userID).Error; err != nil {
        tx.Rollback()
        return err
    }

    var existingProfile Profile
    if err := tx.Where( query: "gorm_user_id = ?", userID).First(&existingProfile).Error; err != nil {
        // create a profile if it doesn't exist
        profile.GormUserID = userID
        if err := tx.Create(&profile).Error; err != nil {
            tx.Rollback()
            return err
        }
    } else {
        // update a profile if it exists
        existingProfile.Bio = profile.Bio
        existingProfile.ProfilePictureURL = profile.ProfilePictureURL
        if err := tx.Save(&existingProfile).Error; err != nil {
            tx.Rollback()
            return err
        }
    }
```

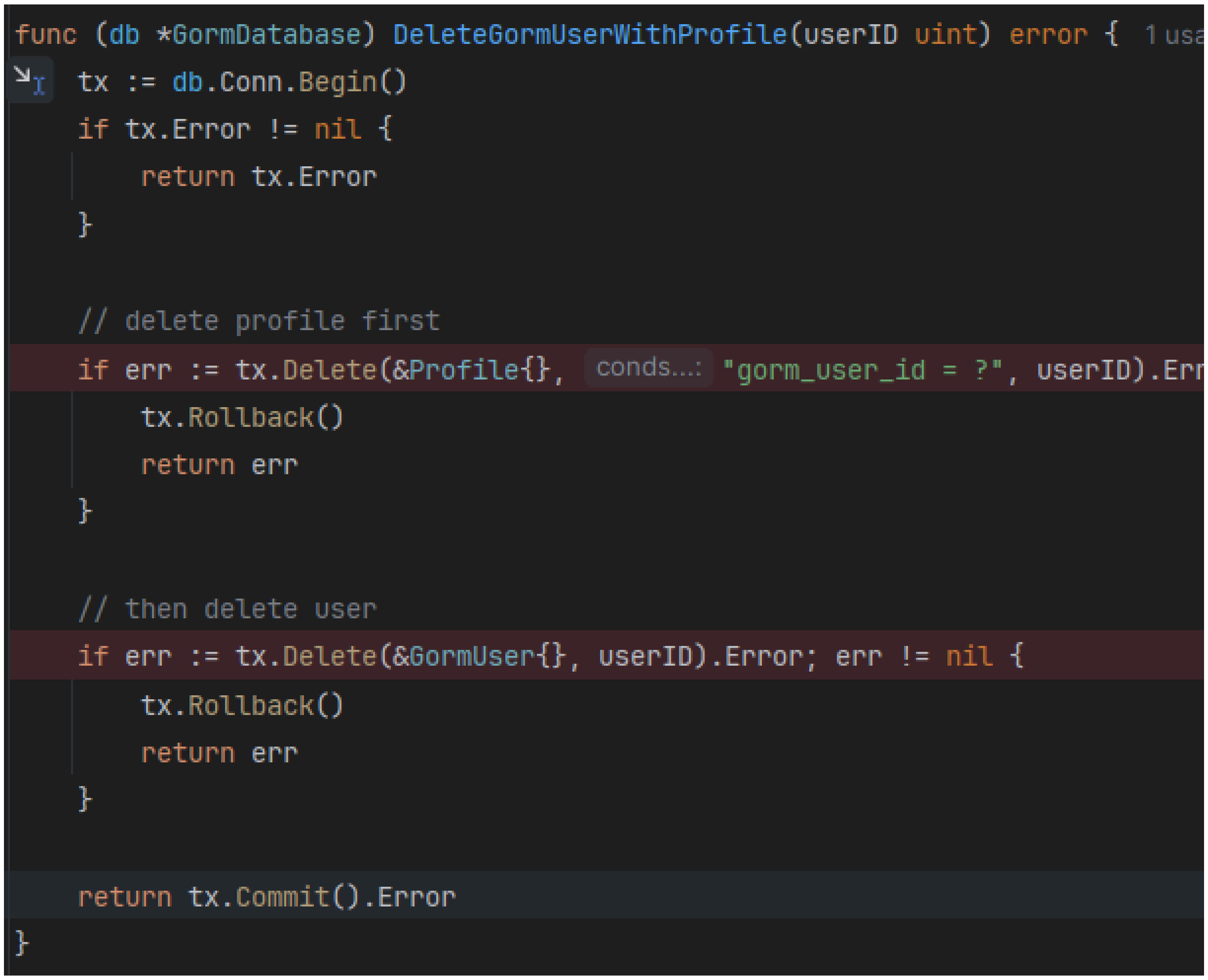


	id	name	age
1	1	Artyom	22

	id	gorm_user_id	bio	profile_picture_url
1	1	1	Senior Software Engi…	http://example.com/artyom_updated.j



```go
// Delete user with profile
if err := dbConn.DeleteGormUserWithProfile( userID: 1); err != nil {
    panic(err)
}
```



```go
func (db *GormDatabase) DeleteGormUserWithProfile(userID uint) error {  1 usa
    tx := db.Conn.Begin()
    if tx.Error != nil {
        return tx.Error
    }

    // delete profile first
    if err := tx.Delete(&Profile{}, conds...: "gorm_user_id = ?", userID).Err
        tx.Rollback()
        return err
    }

    // then delete user
    if err := tx.Delete(&GormUser{}, userID).Error; err != nil {
        tx.Rollback()
        return err
    }

    return tx.Commit().Error
}
```

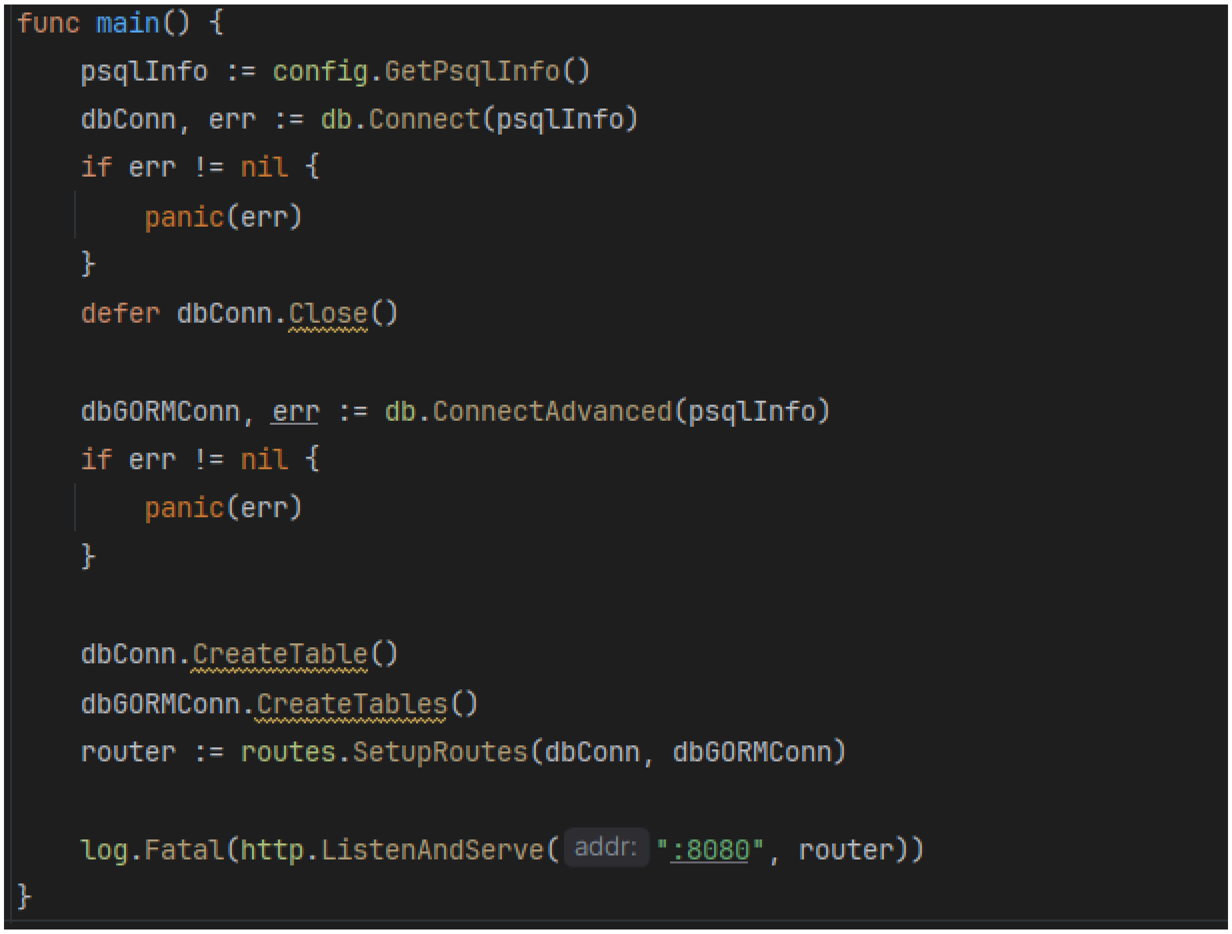
Exercise 3: REST API with Advanced Features

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

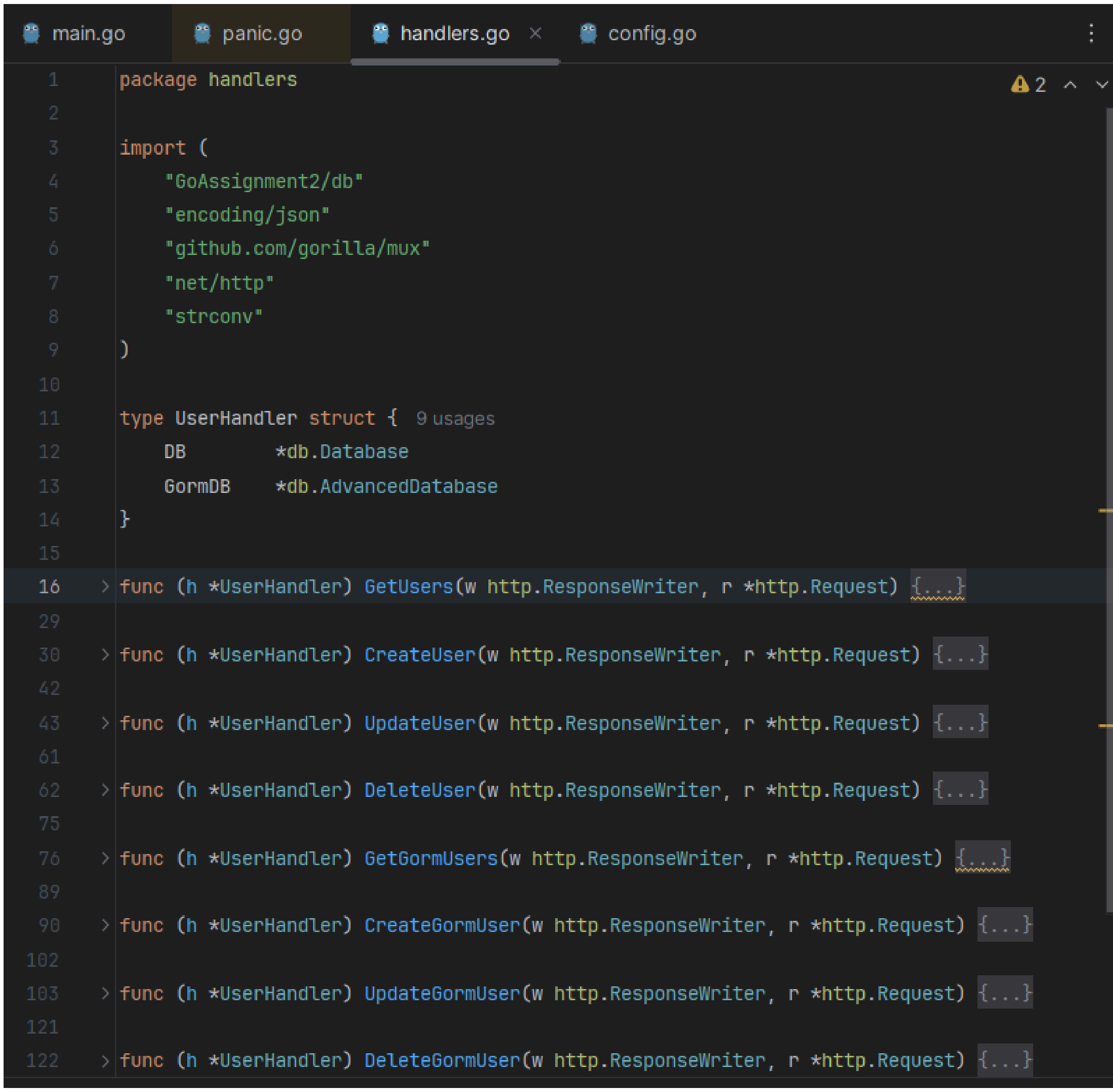
1. **Create REST API Routes with Direct SQL Queries:**
    - **Get Users (GET /users):** Fetch all users with optional query parameters for filtering by `age` and sorting by `name`.
    - **Create User (POST /users):** Insert a new user with validation to ensure `name` is unique.
    - **Update User (PUT /users/{id}):** Update an existing user by ID with validation for `name` uniqueness.
    - **Delete User (DELETE /users/{id}):** Delete a user by ID, ensuring the ID exists.
2. **Create REST API Routes with GORM:**
    - **Get Users (GET /users):** Use GORM to fetch all users with filtering and sorting options.
    - **Create User (POST /users):** Use GORM to insert a new user with validation.

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

3. **Add Pagination and Error Handling:**
   - Implement pagination for the `GET /users` route for both direct SQL and GORM approaches.
   - Add comprehensive error handling for all API endpoints, including validation errors and database errors.
4. **Testing and Documentation:**
   - Write unit tests for each API endpoint.
   - Document the API using Swagger or another API documentation tool.

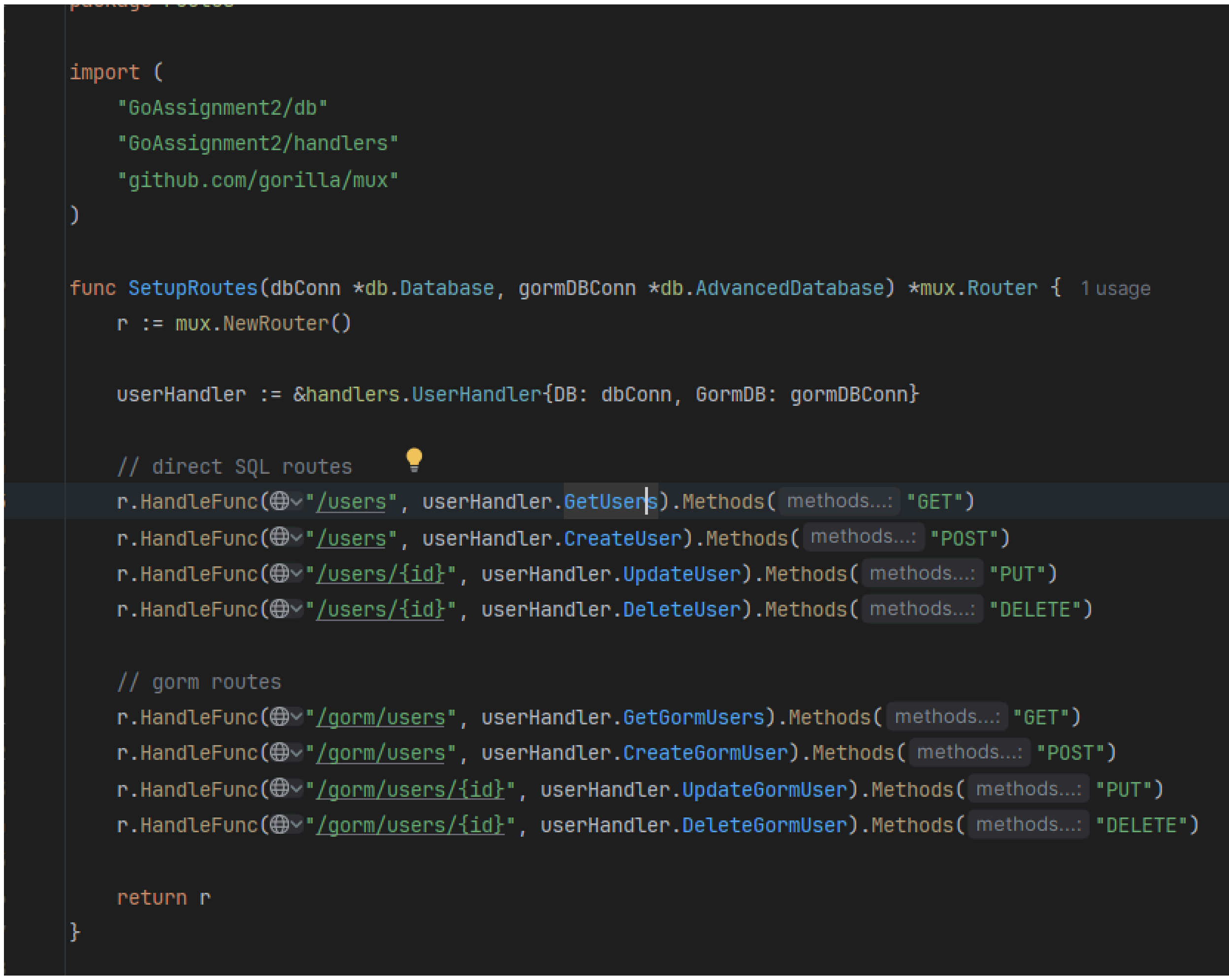


```go
func main() {
    psqlInfo := config.GetPsqlInfo()
    dbConn, err := db.Connect(psqlInfo)
    if err != nil {
        panic(err)
    }
    defer dbConn.Close()

    dbGORMConn, err := db.ConnectAdvanced(psqlInfo)
    if err != nil {
        panic(err)
    }

    dbConn.CreateTable()
    dbGORMConn.CreateTables()
    router := routes.SetupRoutes(dbConn, dbGORMConn)

    log.Fatal(http.ListenAndServe( addr: ":8080", router))
}
```

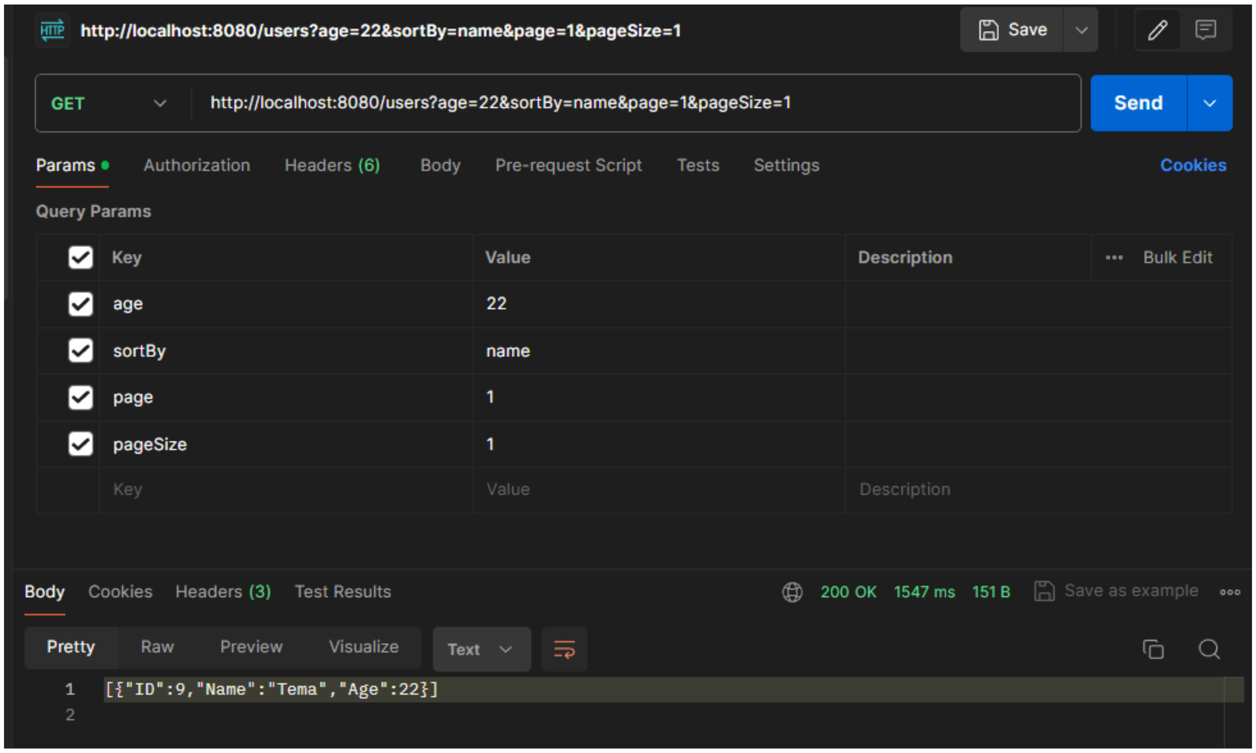


main.go   panic.go   handlers.go   config.go

```go
package handlers

import (
	"GoAssignment2/db"
	"encoding/json"
	"github.com/gorilla/mux"
	"net/http"
	"strconv"
)

type UserHandler struct {  9 usages
	DB       *db.Database
	GormDB   *db.AdvancedDatabase
}

func (h *UserHandler) GetUsers(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) CreateUser(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) UpdateUser(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) DeleteUser(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) GetGormUsers(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) CreateGormUser(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) UpdateGormUser(w http.ResponseWriter, r *http.Request) {...}

func (h *UserHandler) DeleteGormUser(w http.ResponseWriter, r *http.Request) {...}
```



```go
package routes

import (
    "GoAssignment2/db"
    "GoAssignment2/handlers"
    "github.com/gorilla/mux"
)

func SetupRoutes(dbConn *db.Database, gormDBConn *db.AdvancedDatabase) *mux.Router {  1 usage
    r := mux.NewRouter()

    userHandler := &handlers.UserHandler{DB: dbConn, GormDB: gormDBConn}

    // direct SQL routes
    r.HandleFunc("/users", userHandler.GetUsers).Methods( methods...: "GET")
    r.HandleFunc("/users", userHandler.CreateUser).Methods( methods...: "POST")
    r.HandleFunc("/users/{id}", userHandler.UpdateUser).Methods( methods...: "PUT")
    r.HandleFunc("/users/{id}", userHandler.DeleteUser).Methods( methods...: "DELETE")

    // gorm routes
    r.HandleFunc("/gorm/users", userHandler.GetGormUsers).Methods( methods...: "GET")
    r.HandleFunc("/gorm/users", userHandler.CreateGormUser).Methods( methods...: "POST")
    r.HandleFunc("/gorm/users/{id}", userHandler.UpdateGormUser).Methods( methods...: "PUT")
    r.HandleFunc("/gorm/users/{id}", userHandler.DeleteGormUser).Methods( methods...: "DELETE")

    return r
}
```



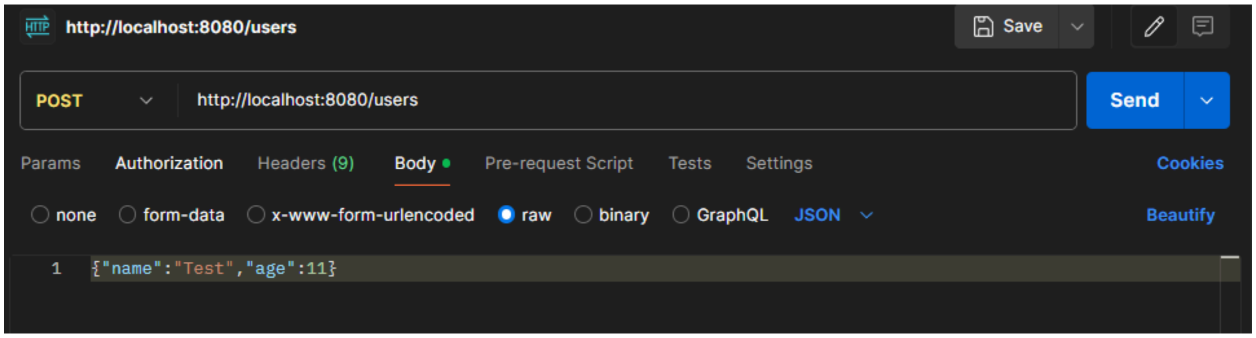
http://localhost:8080/users?age=22&sortBy=name&page=1&pageSize=1

GET http://localhost:8080/users?age=22&sortBy=name&page=1&pageSize=1

Key	Value	Description
age	22	
sortBy	name	
page	1	
pageSize	1	

200 OK 1547 ms 151 B

```
[{"ID":9,"Name":"Tema","Age":22}]
```

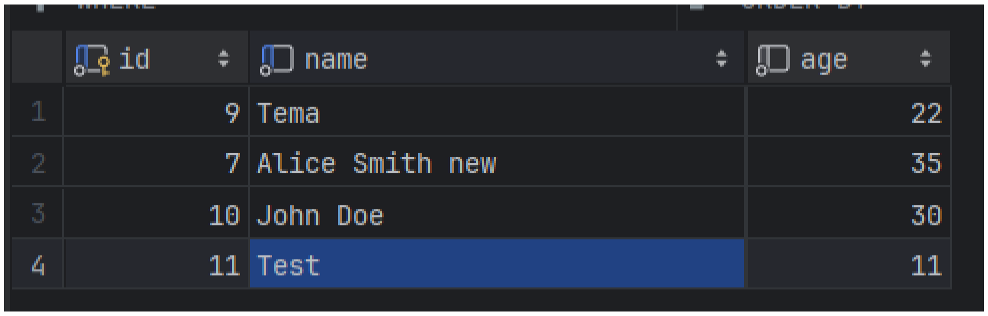


http://localhost:8080/users

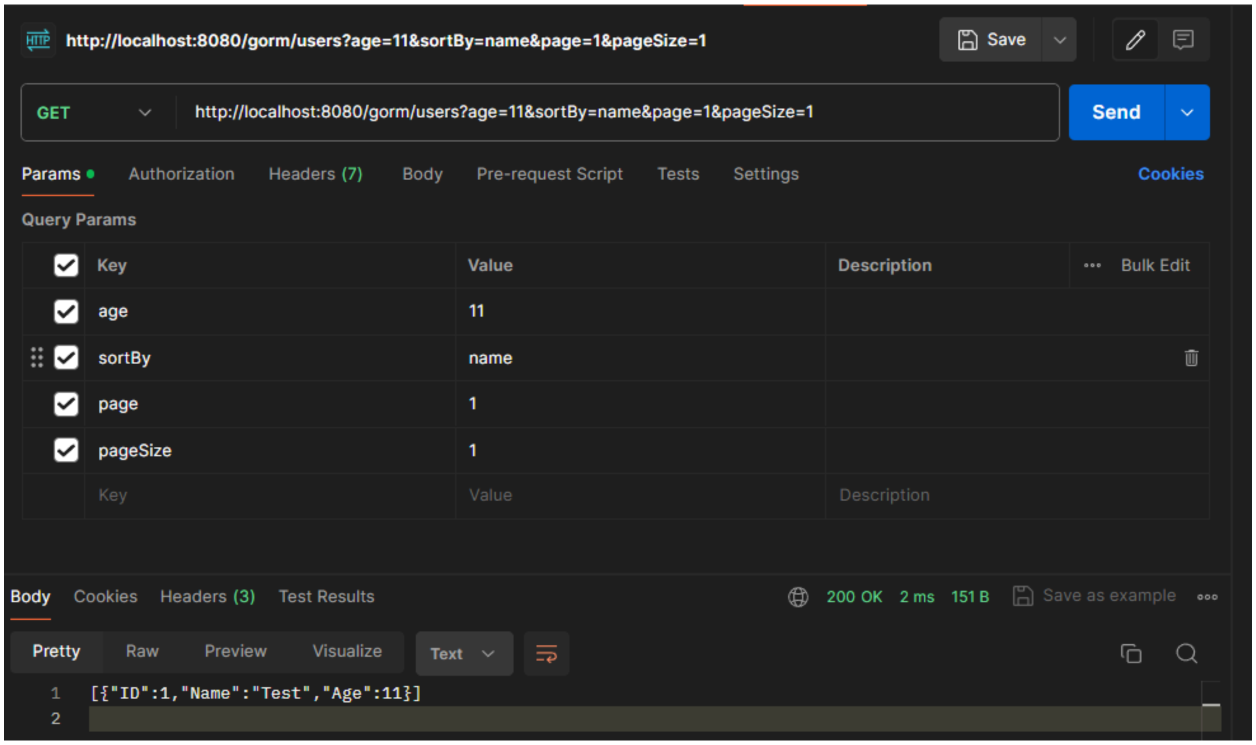
POST http://localhost:8080/users

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{"name":"Test","age":11}
```



	id	name	age
1	9	Tema	22
2	7	Alice Smith new	35
3	10	John Doe	30
4	11	Test	11



http://localhost:8080/gorm/users?age=11&sortBy=name&page=1&pageSize=1

GET http://localhost:8080/gorm/users?age=11&sortBy=name&page=1&pageSize=1

Query Params

Key	Value	Description
age	11	
sortBy	name	
page	1	
pageSize	1	

200 OK 2 ms 151 B

```
[{"ID":1,"Name":"Test","Age":11}]
```