**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.



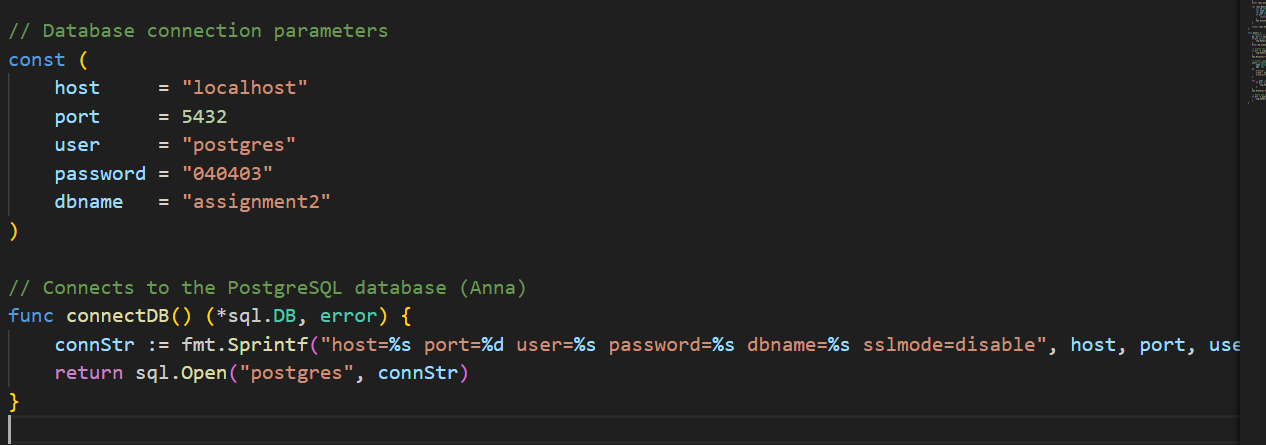
**--Import the necessary packages**

**"database/sql" -** Package sql provides a generic interface around SQL (or SQL-like) databases.

**"fmt" -** Package fmt implements formatted I/O with functions analogous to C's printf and scanf.

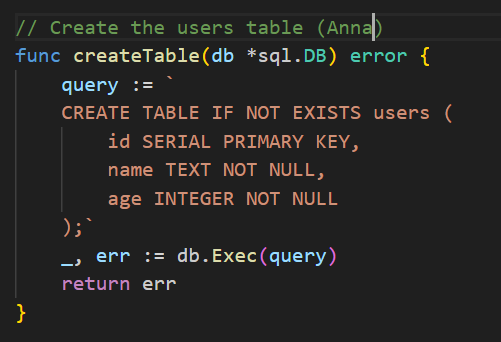
**"log"** - Package log implements a simple logging package. It defines a type, Logger, with methods for formatting output.

**"github.com/lib/pq"** - pq - A pure Go postgres driver for Go's database/sql package

****

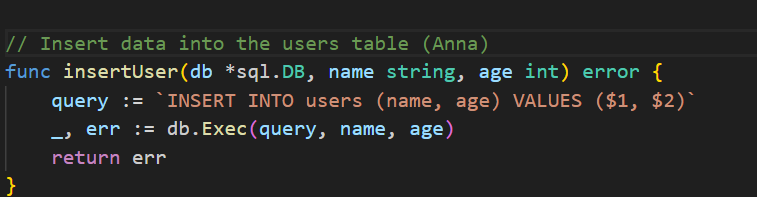
**--Connecting to database using connection parameters**

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



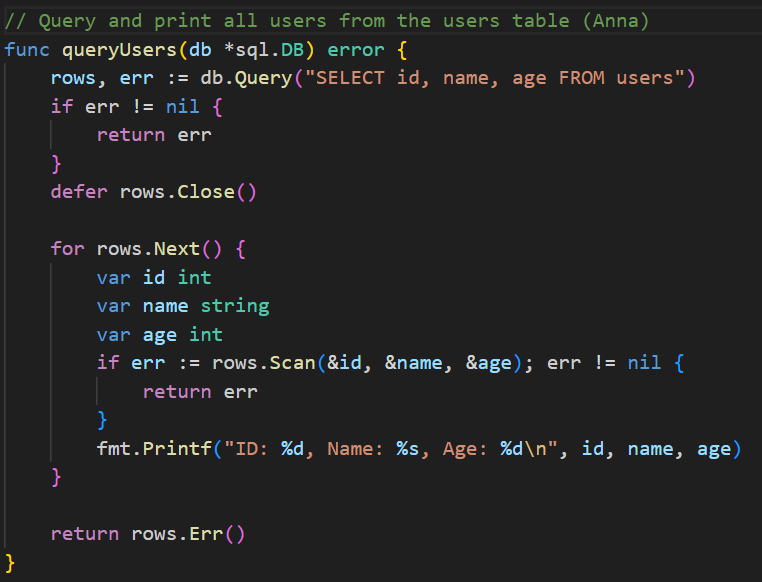
**--Creating a table using query**

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

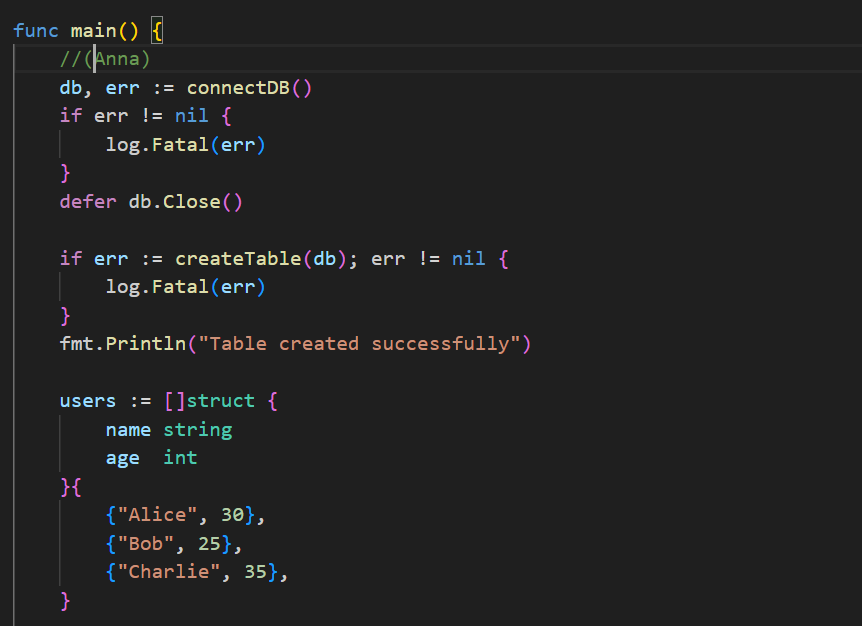


**--Insering data to the table**

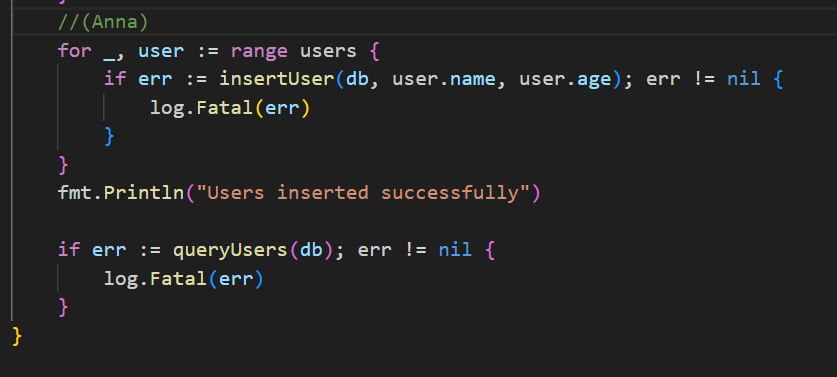
1. **Query Data:** Write a function to query and print all users.



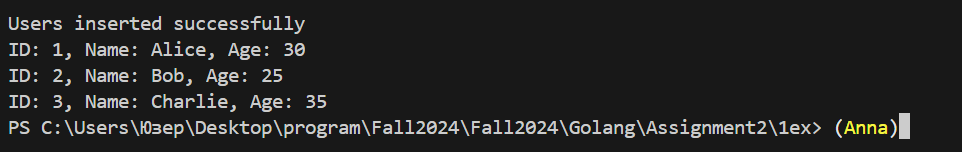
**--Writing a function that prints all users**

****

**--Main function that connects to database, creates table and inserts values**



**--Inserting and printing all users by calling the function**

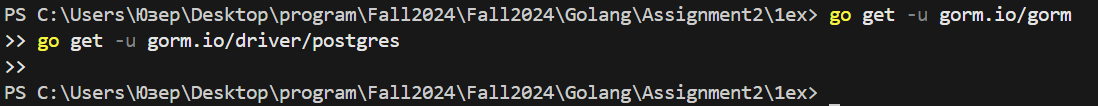


**--Running main.go using “go run main.go”**

**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:



**--Installing drivers using terminal**

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



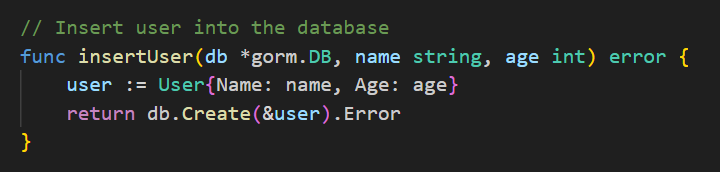
**--Defining the model**

1. **Auto Migrate:** Use GORM’s AutoMigrate to create the users table based on the User struct.



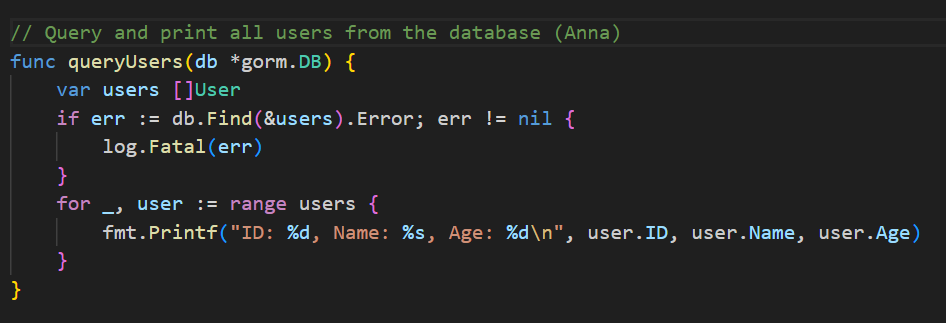
**--Connecting to the database and AutoMigrate**

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

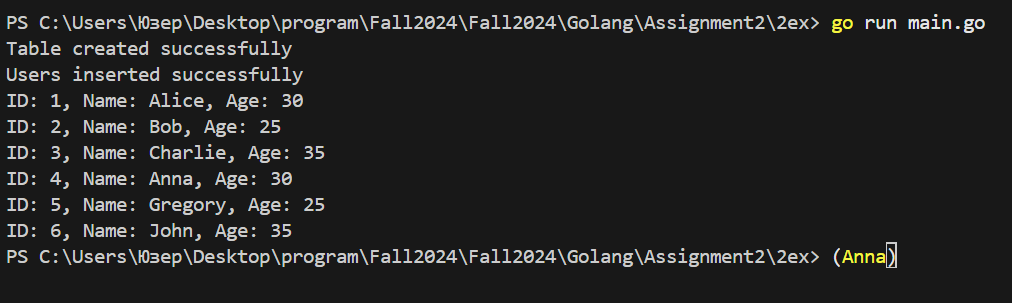


**--Writing a function that inserts data into the database**

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



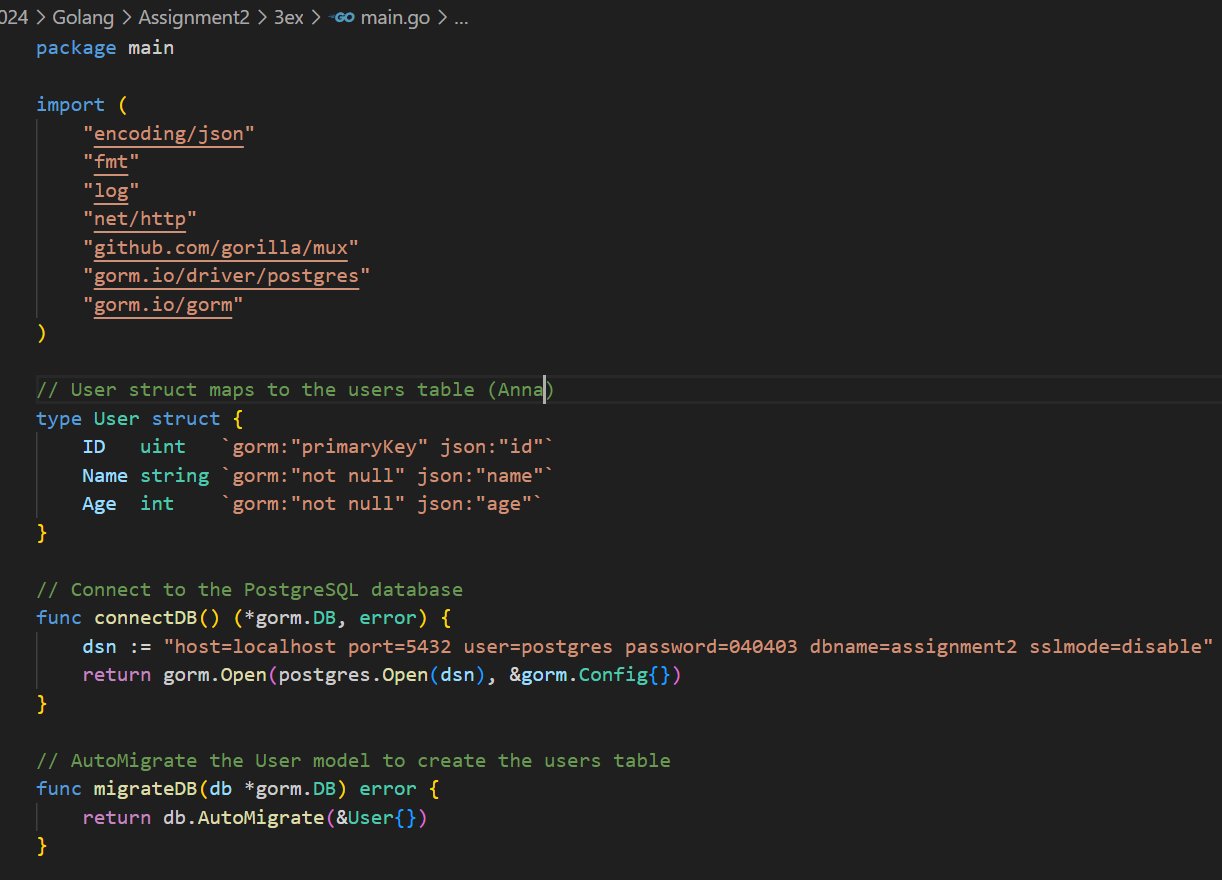
**--Retrieving all users**

****

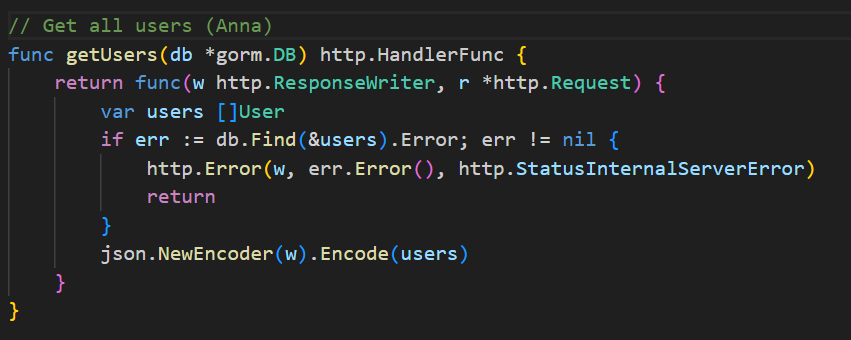
**--Output**

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

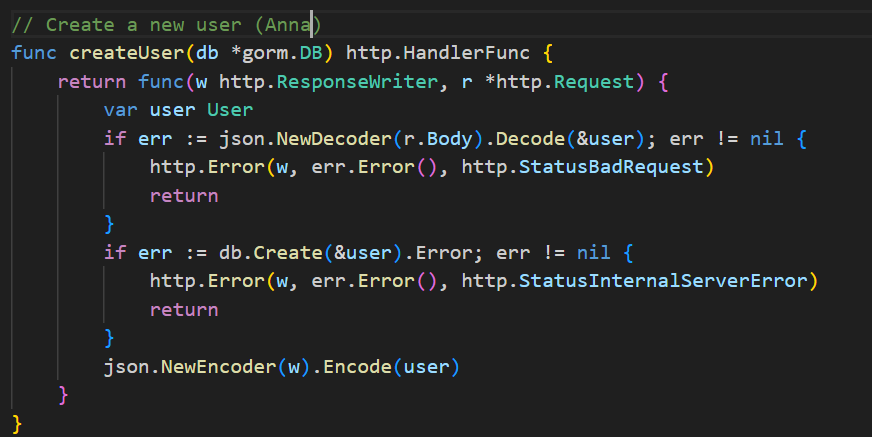
Create a REST API with routes for GET, POST, PUT, and DELETE.



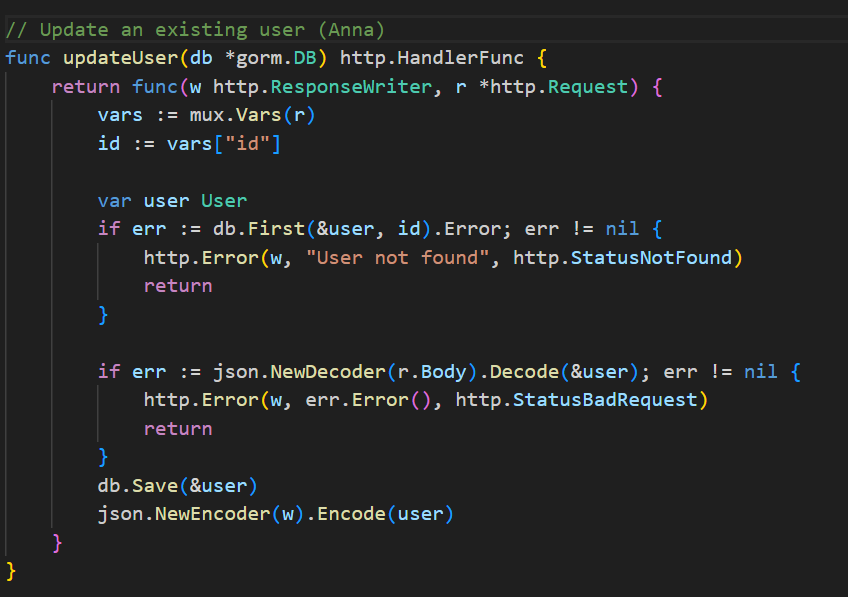
**--Importing packages, creating struct model, connecting to postgres**



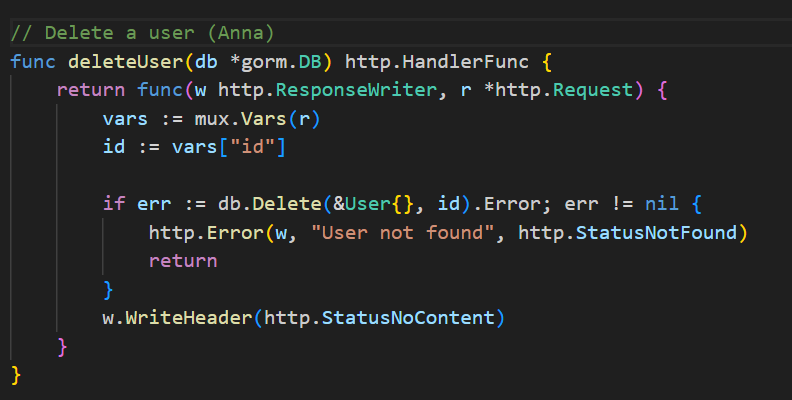
**--Getting Users (GET /users):** A handler to fetch all users from the users table.



**--Creating User (POST /user):** A handler to insert a new user into the users table.



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

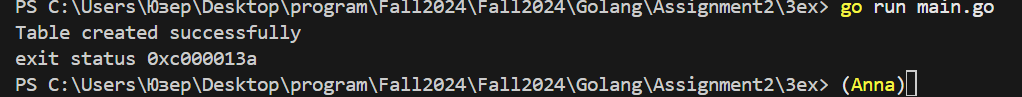


**--Deleting User (DELETE /user/{id}):** A handler to delete a user from the users table.



**--Writing the main function, setting up the router and starting the server**

**http.HandleFunc()** is a function provided by the **net/http** package that allows to associate a specific URL pattern with a handler function.

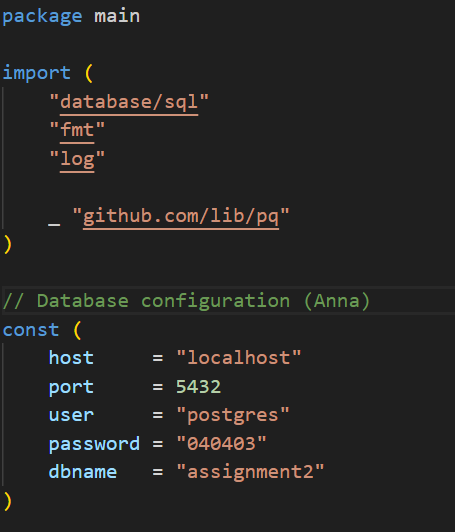


**--Output**

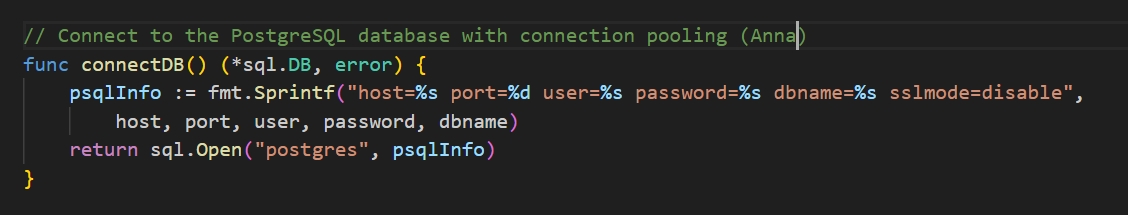
**Exercise 1: Advanced PostgreSQL Operations with database/sql**

**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.

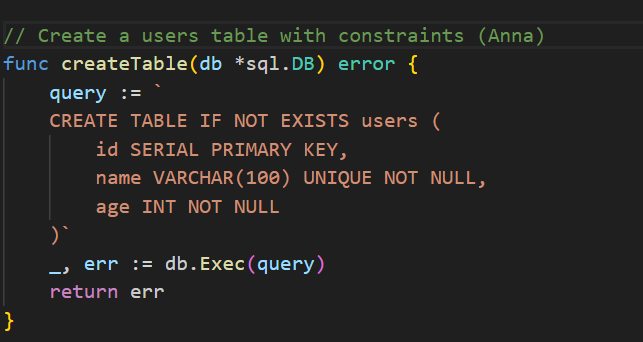


**--Connecting to database**

****

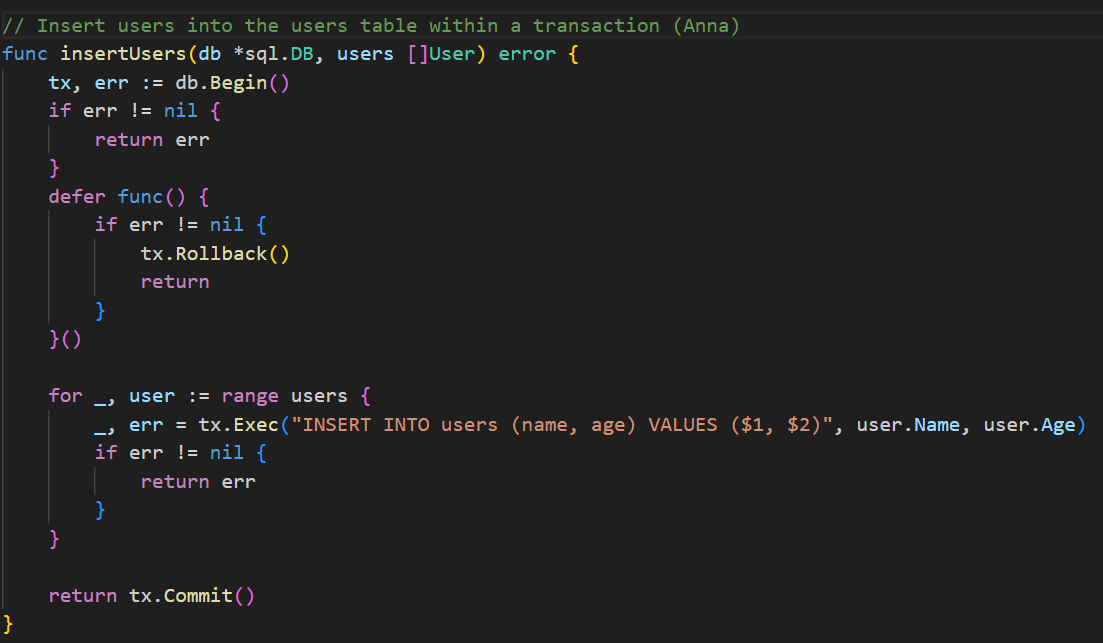
**--Connection pooling**

1. **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.



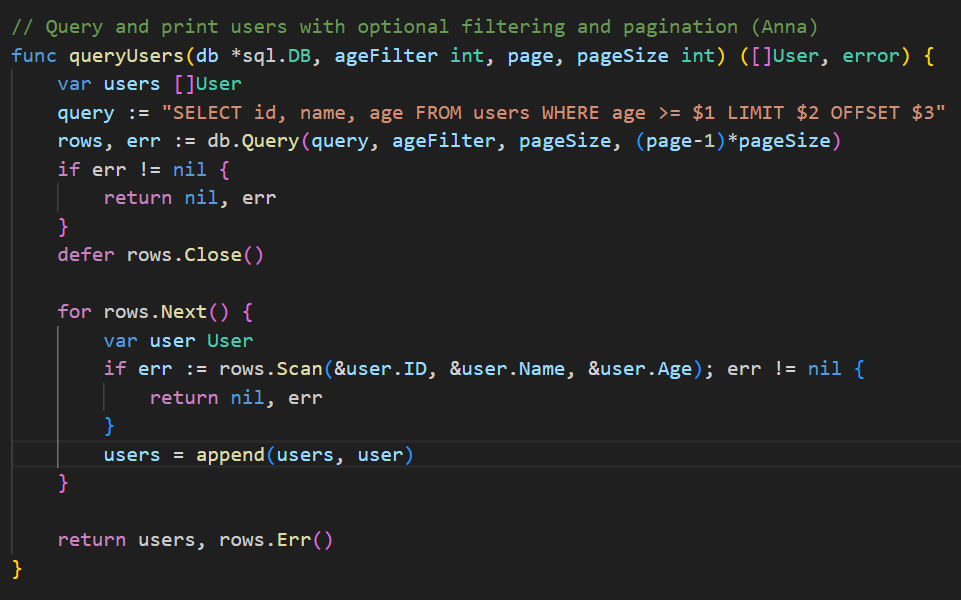
**--Writing a function that creates a table with constraints: id, name, age.**

1. **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.



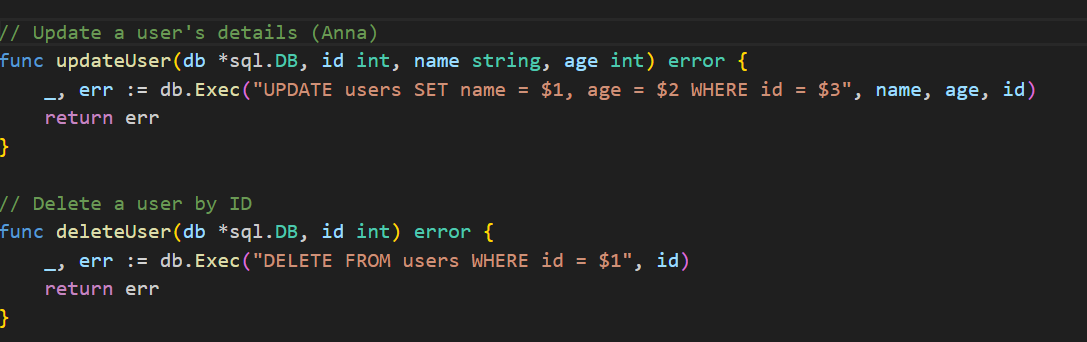
**-- A new transaction is initiated using db.Begin(). A loop iterates over the users array, executing an insert for each user into the users table using tx.Exec(). If all insertions are successful, tx.Commit() is called to save the changes to the database.**

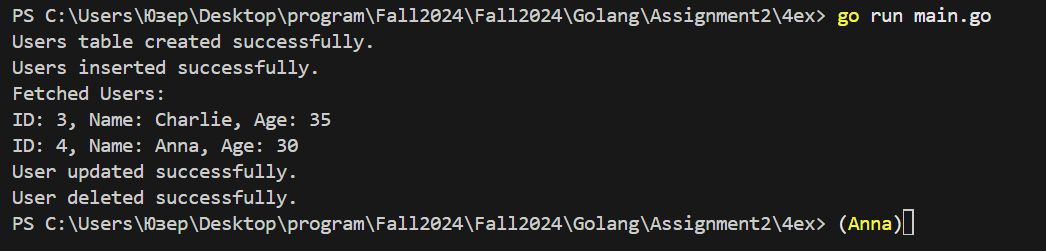
1. **Query Data with Filtering and Pagination:**
   * Write a function to query and print users with optional filters for age and pagination support.
   * Implement pagination to return a specific number of results per page.



**--** **It accepts a database connection (db \*sql.DB), an age filter (ageFilter int), and pagination parameters (page and pageSize). The output is limited by pageSize and offset by (page-1)\*pageSize for pagination. The function returns the slice of users and any error that may have occurred during iteration.**

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





**--Output after running**

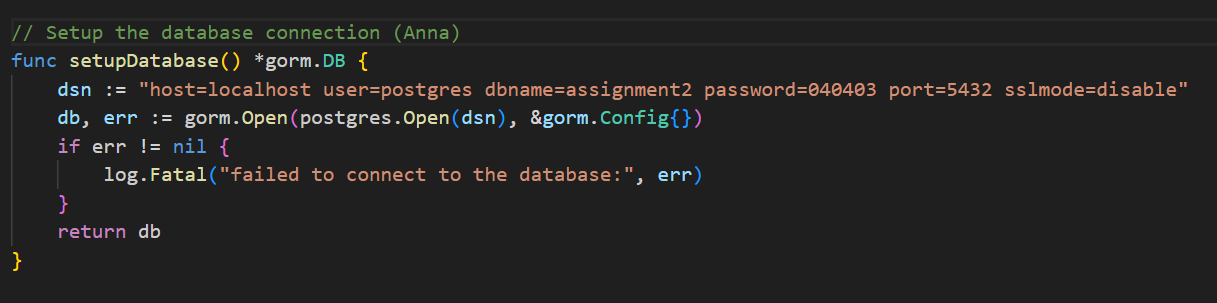
**Exercise 2: Advanced GORM Operations**

**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.

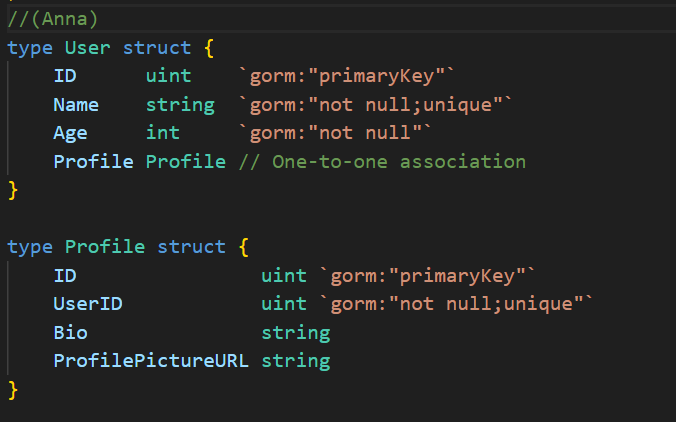


**--Installing GORM and the PostgreSQL driver. Using their packages.**



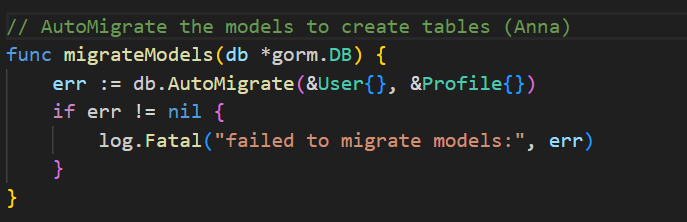
**-- Initializing a connection to the PostgreSQL database**

1. **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.



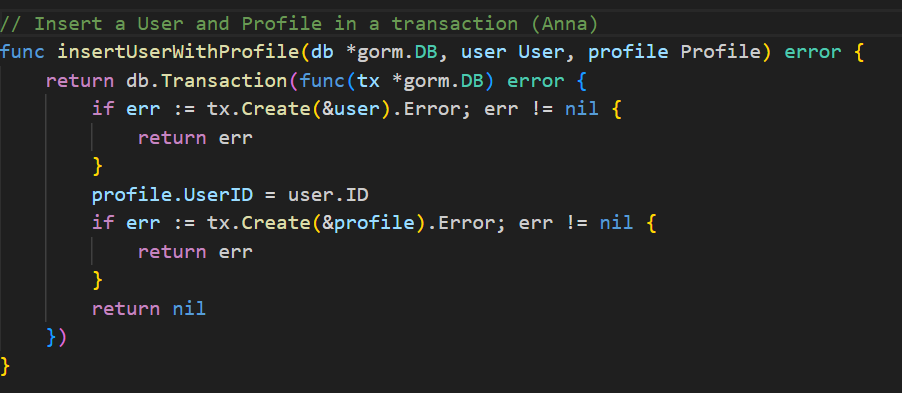
**-- User represents a user in the database with a one-to-one association to Profile. Profile represents a user's profile, associated with a User.**

1. **Auto Migrate with Constraints and Associations:**
   * Use GORM’s AutoMigrate to create tables for User and Profile with appropriate constraints and associations.



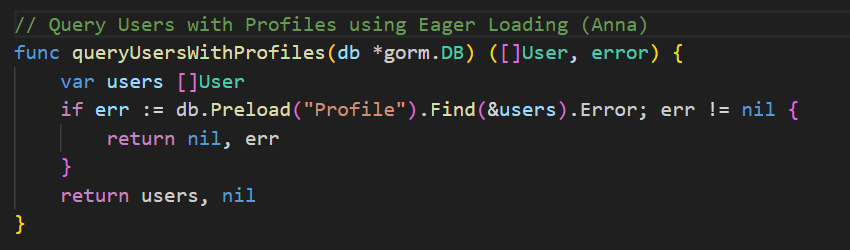
**--Automigration**

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



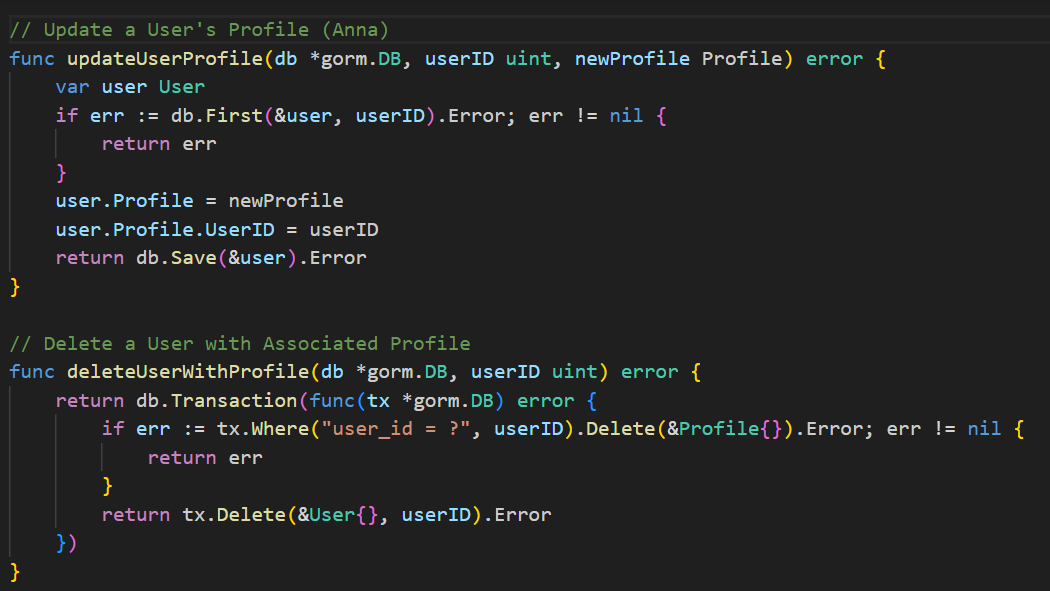
**-- Inserting a User and an associated Profile into the database within a transaction. If the user creation is successful, it assigns the newly created user's ID to the profile and then creates the Profile in the database**

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



**-- The queryUsersWithProfiles function retrieves a list of User records along with their associated Profile records from the database.**

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

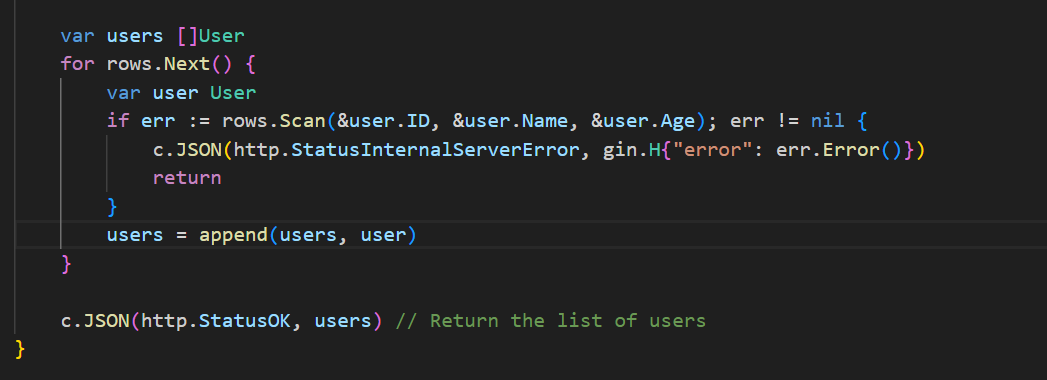


**-- updateUserProfile updates a user's profile with new data, ensuring the user's ID is linked correctly. deleteUserWithProfile deletes a user and their profile in a transactional manner, maintaining data integrity by ensuring both deletions succeed or fail together.**

**Exercise 3: REST API with Advanced Features**

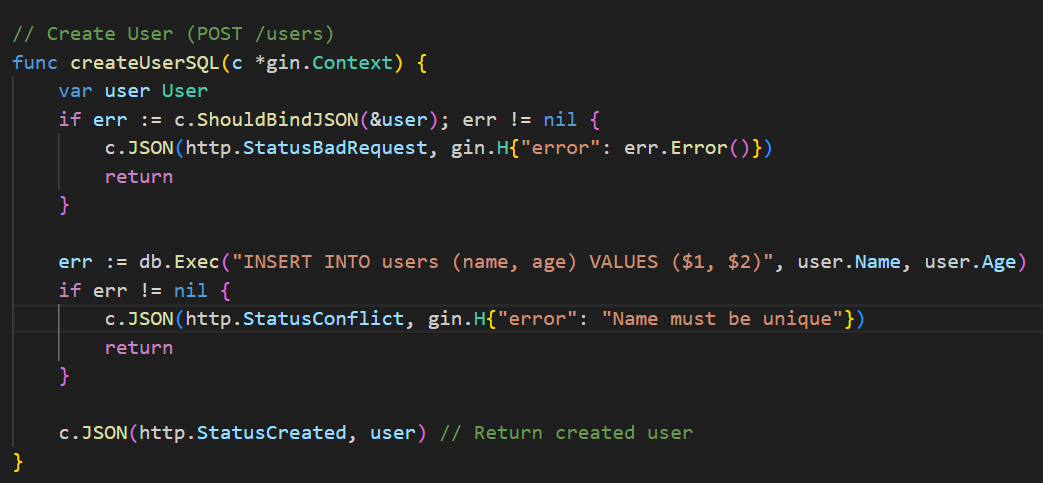
**Objective:** Create a REST API with both direct database/sql queries and GORM, including additional features like filtering and sorting.

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.

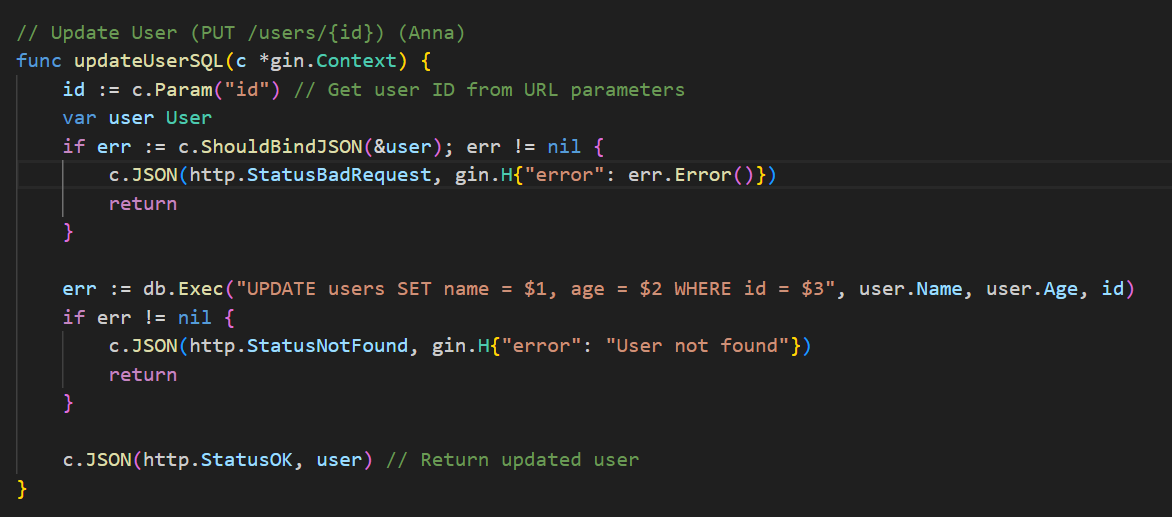
**--This handler fetches all users from the database, with optional filtering by age.**

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



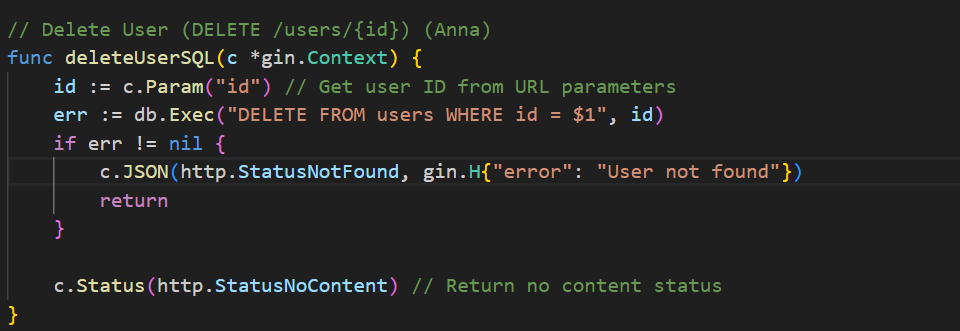
**--Inserting a new user into the database after getting the input.**

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



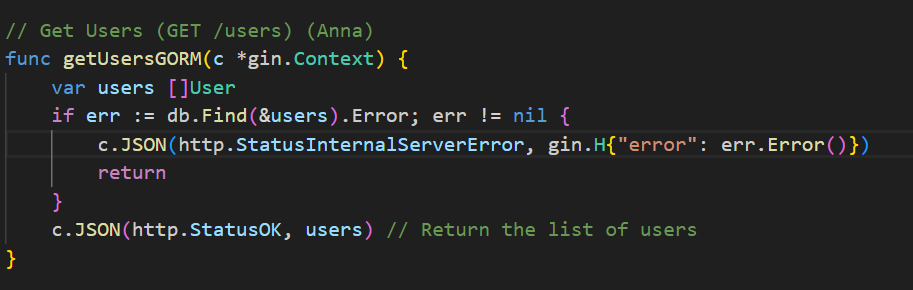
**--Updating an existing user by ID, getting the input.**

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

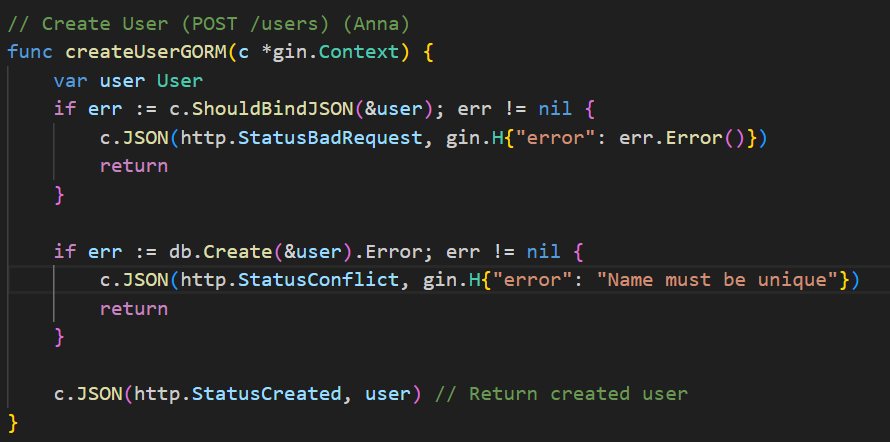


**-- Deleting a user by ID, ensuring the user exists.**

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



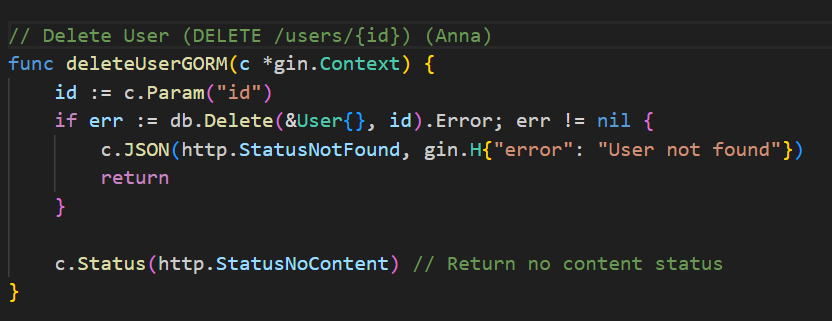
* + **Create User (POST /users):** Use GORM to insert a new user with validation.



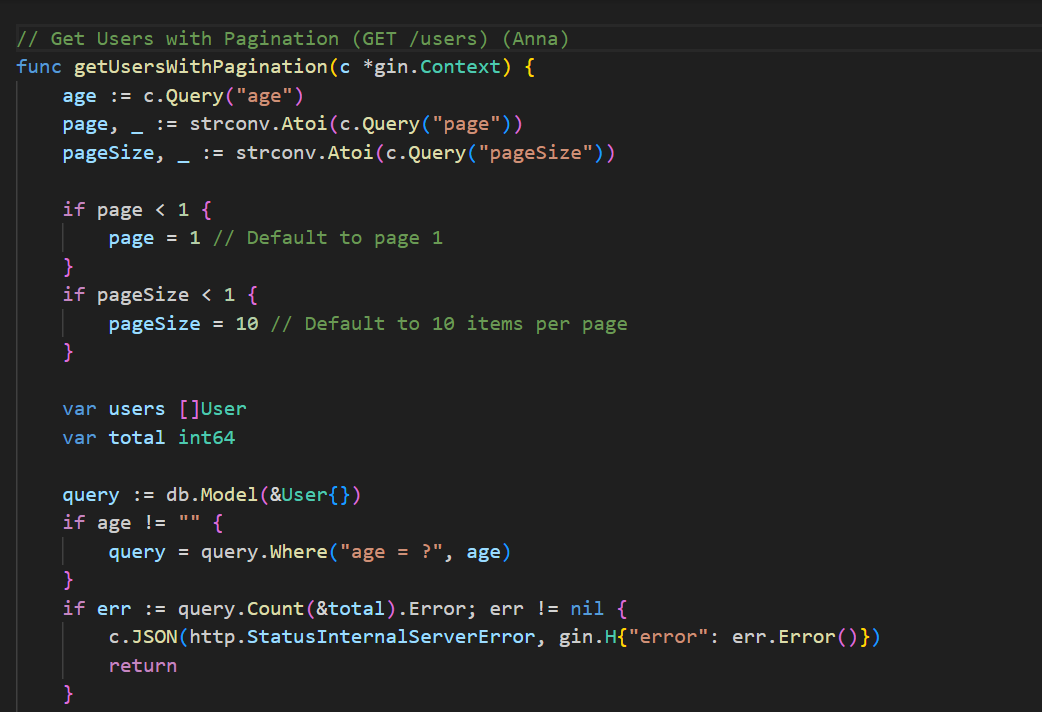
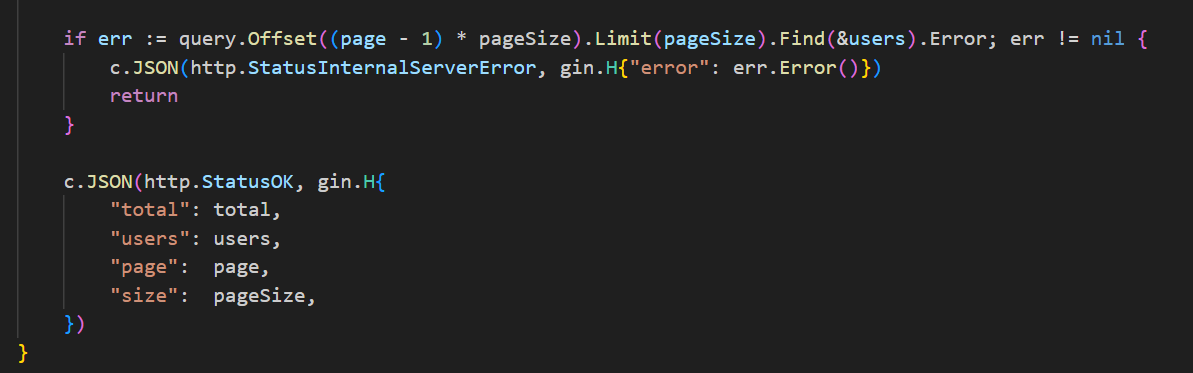
* + **Update User (PUT /users/{id}):** Use GORM to update an existing user by ID.



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

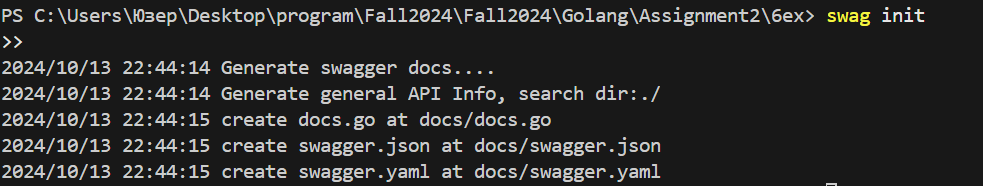


1. **Add Pagination and Error Handling:**
   * Implement pagination for the GET /users route for both direct SQL and GORM approaches.
   * Add comprehensive error handling for all API endpoints, including validation errors and database errors.

**--Fetch all users with optional query parameters for filtering by age and pagination.**

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



***Summary***

In Assignment 2, I learned how to use GORM with PostgreSQL to manage user and profile data, focusing on creating, retrieving, updating, and deleting records. I also built a REST API for these operations, highlighting the importance of error handling and documentation.