Trabajo Práctico 2 - Persistencia de Datos en Go

Ejercicio 1: "SQL Directo" con `database/sql`

Objetivo: Implementar la capa de persistencia utilizando el paquete estándar `database/sql` de Go para interactuar con una base de datos PostgreSQL.

1. **Configuración de la Base de Datos**:
   - Inicia una instancia de PostgreSQL usando Docker.
   - Crea una base de datos y una tabla `users` con la siguiente estructura:
     ```sql
     CREATE TABLE users (
         id SERIAL PRIMARY KEY,
         name VARCHAR(255) NOT NULL,
         email VARCHAR(255) UNIQUE NOT NULL,
         created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
     );
     ```
2. **Capa de Datos en Go**:
   - Crea un nuevo proyecto en Go.
   - Define un `struct User` que corresponda a la tabla de la base de datos.
   - Implementa un repositorio (`UserRepository`) que encapsule la lógica de acceso a datos.
   - Este repositorio debe tener un `*sql.DB` y métodos para las operaciones CRUD:
     - `CreateUser(user *User) error`
     - `GetUserByID(id int) (*User, error)`
     - `ListUsers() ([]*User, error)`
     - `UpdateUser(user *User) error`
     - `DeleteUser(id int) error`
   - Asegúrate de manejar correctamente las conexiones, los errores (ej. `sql.ErrNoRows`) y de usar consultas parametrizadas para prevenir inyecciones SQL.
3. **Verificación con Pruebas Unitarias**:
   - Crea un archivo `user_repository_test.go`.
   - Define una función de prueba como `TestUserRepository_CRUD` que reciba un `*testing.T`.
   - Dentro de la prueba, establece una conexión a una base de datos de prueba (puede ser la misma, pero es recomendable usar una base de datos separada para pruebas).
   - Instancia tu `UserRepository`.
   - Escribe un flujo de prueba que verifique el ciclo completo de CRUD:
     1. **Crear**: Llama a `CreateUser` y verifica que no haya errores.
     2. **Leer**: Llama a `GetUserByID` con el ID del usuario creado y comprueba que los datos son correctos.
     3. **Actualizar**: Llama a `UpdateUser` para cambiar algún dato y luego usa `GetUserByID` de nuevo para asegurar que el cambio se aplicó.
     4. **Listar**: Llama a `ListUsers` y verifica que el usuario creado (y actualizado) esté en la lista.
     5. **Eliminar**: Llama a `DeleteUser` y luego intenta obtener el usuario por su ID, esperando un error (ej. `sql.ErrNoRows`) para confirmar que fue eliminado.
   - Utiliza las funciones del paquete `testing` como `t.Run()`, `t.Errorf()` o `t.Fatalf()` para estructurar tus pruebas y reportar fallos.

Ejercicio 2: Generación de Código con `sqlc`

Objetivo: Utilizar `sqlc` para generar código Go seguro y tipado a partir de consultas SQL, eliminando la necesidad de escribir código de acceso a datos manually.

1. **Configuración de `sqlc`**:
   - Instala `sqlc`.
   - Crea un archivo de configuración `sqlc.yaml` para tu proyecto, especificando el motor (`postgresql`), la ubicación del esquema y las consultas.
2. **Definición de Esquema y Consultas**:
   - Crea un directorio `db/schema` y dentro un archivo `schema.sql` con la definición de la tabla `users` del ejercicio anterior.
   - Crea un directorio `db/queries` y dentro un archivo `users.sql`. En este archivo, escribe las consultas SQL para las operaciones CRUD, utilizando las anotaciones de `sqlc`:
     ```sql
     -- name: GetUser :one
     SELECT * FROM users WHERE id = $1;
     ```

```sql
-- name: ListUsers :many
SELECT * FROM users ORDER BY name;

-- name: CreateUser :one
INSERT INTO users (name, email) VALUES ($1, $2) RETURNING *;

-- name: UpdateUser :exec
UPDATE users SET name = $2, email = $3 WHERE id = $1;

-- name: DeleteUser :exec
DELETE FROM users WHERE id = $1;
```

3. **Generación y Uso del Código**:
   - Ejecuta `sqlc generate` en la raíz de tu proyecto.
   - Verifica que se haya generado un nuevo paquete Go con las funciones y structs correspondientes.
   - Crea un `main.go` que utilice el código generado por `sqlc` para realizar las mismas operaciones CRUD del ejercicio 1. Compara la simplicidad y seguridad de este enfoque con el de "SQL Directo".
4. **Verificación con Pruebas Unitarias**:
   - Crea un archivo de prueba para el código generado, por ejemplo, `db_test.go`.
   - Define una función de prueba `TestQueries_CRUD` que reciba un `*testing.T`.
   - Dentro de la prueba, establece una conexión a la base de datos y crea una instancia del objeto `*db.Queries` generado por `sqlc`.
   - Escribe pruebas para cada una de las operaciones generadas, siguiendo un flujo lógico:
     1. **CreateUser**: Llama al método, pasando un `context.Context` y los parámetros necesarios. Verifica que el usuario retornado sea correcto.
     2. **GetUser**: Llama al método con el ID del usuario recién creado y comprueba que los datos coinciden.
     3. **UpdateUser**: Llama al método para actualizar el usuario.
     4. **ListUsers**: Verifica que la lista de usuarios contenga el usuario actualizado.
     5. **DeleteUser**: Elimina el usuario y verifica que ya no se pueda obtener.
   - Compara la simplicidad de probar este código en comparación con el del Ejercicio 1. El código generado por `sqlc` es más predecible y requiere menos "mocking" o configuración compleja.

Ejercicio 3: Explorando un ORM con GORM

Objetivo: Comprender el funcionamiento básico de un ORM (Object-Relational Mapper) en Go utilizando GORM para abstraer completamente la base de datos.

1. **Configuración de GORM**:
   - En un nuevo proyecto Go, agrega GORM y el driver de PostgreSQL como dependencias.
   - Establece una conexión a la base de datos usando GORM.
2. **Modelo y Migración**:
   - Define un `struct User` utilizando las etiquetas (`tags`) de GORM para definir el modelo.

```go
import "gorm.io/gorm"

type User struct {
    gorm.Model
    Name  string
    Email string `gorm:"unique"`
}
```

   - Utiliza la función `db.AutoMigrate(&User{})` de GORM para que cree o actualice automáticamente la tabla en la base de datos.
3. **Operaciones CRUD con GORM**:
   - Implementa las operaciones CRUD utilizando los métodos de GORM:
     - Crear: `db.Create(&User{...})`
     - Leer (uno): `db.First(&user, 1)`
     - Leer (todos): `db.Find(&users)`
     - Actualizar: `db.Model(&user).Update("name", "new_name")`
     - Eliminar: `db.Delete(&User{}, 1)`
   - Reflexiona sobre el nivel de abstracción: ¿Qué tan diferente es interactuar con la base de datos a través de objetos y métodos en comparación con `sqlc` o SQL directo?
4. **Verificación con Pruebas Unitarias**:
   - Crea un archivo de prueba, por ejemplo, `gorm_test.go`.

- Define una función de prueba `TestGORM_CRUD` que reciba un `*testing.T`.
- Dentro de la prueba, configura una conexión a una base de datos de prueba con GORM y ejecuta `AutoMigrate`. Es una buena práctica limpiar la tabla antes de cada ejecución de prueba para asegurar un estado inicial limpio.
- Escribe un flujo de prueba para las operaciones CRUD con GORM:
  1. **Crear**: Crea un nuevo `User` y usa `db.Create()`. Comprueba que `result.Error` sea `nil` y que el `ID` del objeto se haya actualizado.
  2. **Leer**: Usa `db.First()` para buscar el usuario por su `ID` y verifica que los datos recuperados sean correctos.
  3. **Actualizar**: Modifica el objeto `User` y usa `db.Model().Update()` o `db.Save()` para persistir los cambios. Vuelve a leer el registro para confirmar la actualización.
  4. **Eliminar**: Usa `db.Delete()` para borrar el usuario. Intenta buscarlo de nuevo y espera un error como `gorm.ErrRecordNotFound`.

Recursos Adicionales

- Paquete `database/sql`[1]
- Driver `pq` para PostgreSQL[2]
- Documentación de `sqlc`[3]
- Documentación de GORM[4]

Trabajo de Cursada: Persistiendo el Dominio

Objetivo: Definir la estructura de datos de tu aplicación y preparar la capa de acceso a la base de datos.

Continuando con la aplicación que definiste en el TP1, ahora crearás la base para almacenar tus datos de forma permanente.

1. **Diseño de la Base de Datos**:
   - Basado en el dominio que elegiste, define la estructura de la tabla principal en SQL. Por ejemplo, si es una lista de tareas, la tabla `tasks` podría tener columnas como `id`, `title`, `description`, y `completed`.
   - Crea un archivo `schema.sql` con la sentencia `CREATE TABLE` para tu entidad.
2. **Definición de Consultas**:
   - Crea un archivo `queries.sql` donde escribirás las consultas SQL para las operaciones CRUD básicas:
     - Una consulta para crear un nuevo registro (`Create...`).
     - Una consulta para obtener un registro por su ID (`Get...`).
     - Una consulta para listar todos los registros (`List...`).
     - Una consulta para actualizar un registro (`Update...`).
     - Una consulta para borrar un registro (`Delete...`).
   - Utiliza las anotaciones que requiere `sqlc` (`-- name: ...`).
3. **Generación del Código de Acceso a Datos**:
   - Configura `sqlc.yaml` para que apunte a tus archivos de esquema y consultas.
   - Ejecuta `sqlc generate` para que genere el código Go para interactuar con tu base de datos.
   - En esta etapa, no es necesario conectar la capa de datos a un servidor web. El objetivo es tener el paquete de base de datos listo para ser usado en el siguiente práctico.

[1] https://pkg.go.dev/database/sql
[2] https://pkg.go.dev/github.com/lib/pq
[3] https://sqlc.dev/
[4] https://gorm.io/