# INTRODUCCIÓN A LA CLASE

STORAGE IMPLEMENTATION

# Objetivos de esta clase

- Continuar con la implementación de la interfaz de Repository.
- Aplicar joins en queries de BD.
- Implementar context en el Repository.
- Entender el uso del context.

# IMPLEMENTACIÓN GetAll

STORAGE IMPLEMENTATION

# GetAll

```go
func (r *repository) GetAll() ([]models.Product, error) {
    var products []models.Product
    db := db.StorageDB
    rows, err := db.Query("SELECT * FROM products")
    if err != nil {
        log.Println(err)
        return nil, err
    }
    // se recorren todas las filas
    for rows.Next() {
        // por cada fila se obtiene un objeto del tipo Product
        var product models.Product
        if err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count, &product.Price); err != nil {
            log.Fatal(err)
            return nil, err
        }
        //se añade el objeto obtenido al slice products
        products = append(products, product)
    }
    return products, nil
}
```

# GetAll

El bloque de código anterior es la implementación del método GetAll, el cual permite obtener todos los productos existentes en la tabla. Los dos métodos elementales usados para interactuar con la base en este caso son:

- db.Query: Establece la consulta a ejecutar
- stmt.Exec: Ejecuta la consulta, la cual en este caso no se acompaña con ningun parametro adicional.
- Con rows.Next() se recorren o se itera sobre todos los resultados obtenidos.
- rows.Scan() permite copiar cada campo sobre nuestro struct.

# IMPLEMENTACIÓN Delete

STORAGE IMPLEMENTATION

# Delete

```
{}
```

```go
func (r *repository) Delete(id int) error {
    db := db.StorageDB                                          // se inicializa la base
    stmt, err := db.Prepare("DELETE FROM products WHERE id = ?") // se prepara la sentencia SQL a ejecutar
    if err != nil {
        log.Fatal(err)
    }
    defer stmt.Close()     // se cierra la sentencia al terminar. Si quedan abiertas se genera consumos de memoria

    _, err = stmt.Exec(id) // retorna un sql.Result y un error

    if err != nil {
        return err
    }

    return nil
}
```

# Delete

El Delete se ejecuta haciendo uso de:

- db.Prepare: Devuelve el statement o sentencia lista para incorporar valores y ejecutar.

- stmt.Exec: Este método pertenece al objeto statement y ejecuta la query preparada con los parámetros que recibe de entrada

# IMPLEMENTANDO JOINS

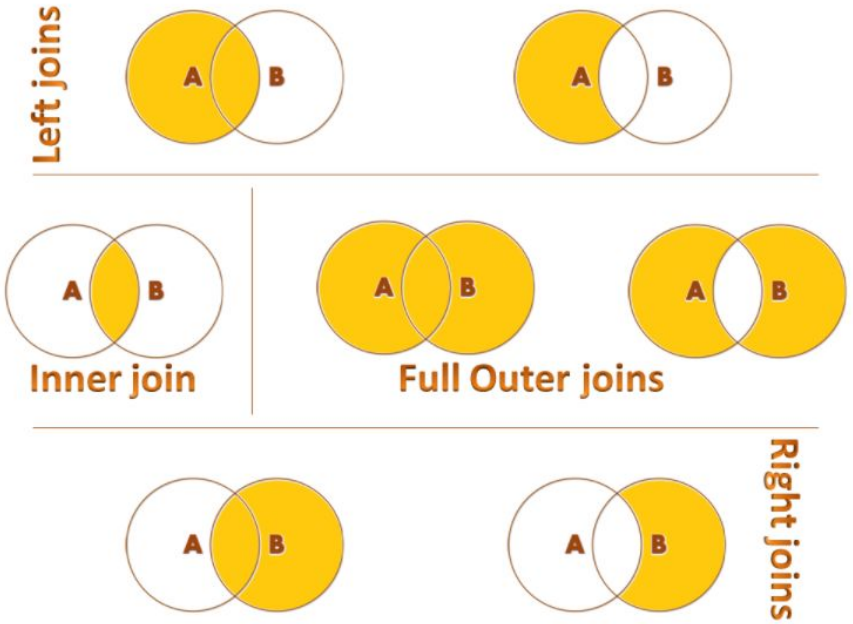
STORAGE IMPLEMENTATION

# Joins

En ocasiones, se presenta la necesidad de ejecutar consultas(queries) en las que se requiere unir la información de tablas distintas. JOIN es la cláusula que se usa en la instrucción SELECT; que permite combinar dos tablas en un solo resultado, basándose en uno o más campos que comparten (campos llave). Supongamos que en la tabla products, existe un campo llamado “id_warehouse”, que relaciona el producto con los datos de la tabla de warehouse. Si requerimos “cruzar” o “combinar” los datos de ambas tablas relacionados correctamente, debemos hacer uso de los Join query.

# Tipos de Joins

La infografía muestra los distintos tipos de JOINS. A efectos de esta clase vamos a trabajar con el más común y de mayor uso que es el INNER JOIN



# Inner Join

Utilizaremos la premisa enunciada anteriormente en la que existe una tabla llamada “warehouses”, que contiene el nombre y dirección de distintos almacenes. Mientras que en la tabla products, existe un nuevo campo llamado “id_warehouse” que relaciona cada producto con un almacén. Así es el Inner Join que nos permite combinar estas dos tablas :

```
{} SELECT * FROM products INNER JOIN warehouses ON products.id_warehouse = warehouses.id;
```

Con este Join, ya podemos implementar un método en el Repository llamado GetFullData() que retorne un struct con los datos del producto y también del almacén donde se encuentra.

# IMPLEMENTACIÓN GetFullData

STORAGE IMPLEMENTATION

# GetFullData

```
{}
```

```go
func (r *repository) GetFullData(id int) models.Product {
    var product models.Product
    db := db.StorageDB
    innerJoin := "SELECT products.id, products.name, products.type, products.count, products.price, warehouses.name, warehouses.adress " +
        "FROM products INNER JOIN warehouses ON products.id_warehouse = warehouses.id " +
        "WHERE products.id = ?"
    rows, err := db.Query(innerJoin, id)
    if err != nil {
        log.Println(err)
        return product
    }
    for rows.Next() {
        if err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count, &product.Price, &product.Warehouse,
                            &product.WarehouseAdress); err != nil {
            log.Fatal(err)
            return product
        }
    }
    return product
}
```

# GetFullData

El método se implementó de forma análoga al método GetOne. Sin embargo, es necesario alertar los siguiente:

- La query del inner Join especifica exactamente qué campos requerimos para evitar errores en durante el rows.Scan.

- En las implementaciones anteriores veníamos haciendo un Select All (select *) de la tabla, esto es una mala práctica. Porque cuando se añadan nuevos campos a la tabla se generan errores durante rows.Scan. Debido a que la consulta trae más campos de lo esperado. Lo correcto es especificar el nombre de cada campo requerido en todas las query o consultas a ejecutar, sin importar si es o no un join

# CONTEXT

STORAGE IMPLEMENTATION

## // ¿Qué es Context?

Context brinda la posibilidad de efectuar cancelaciones de las queries mientras están en plena ejecución. Esto permite administrar los tiempos de duración y la aplicación de timeouts a las consultas a la DB.

# Utilidad del Context

Hay distintos escenarios en los que es vital contar con el context:

- Demora en la Consulta: Cuando una consulta demora más de lo habitual, el context puede cancelar la ejecución de la misma tras un tiempo determinado. Esto libera la conexión en curso y la asigna de nuevo al pool de conexiones disponibles. También permite devolver al cliente una respuesta oportuna. Es preferible retornar un error oportunamente, que hacer esperar al cliente por una respuesta que probablemente también sea errónea.
- Cuando un cliente cancela un request. Ejemplo: Un usuario cierra la aplicación mobile. Ya no tiene sentido retornar una respuesta por lo que el context cancela la consulta y libera los recursos.

# ¿Dónde definir el Context?

En algunos casos, el timeout sobre una consulta o conexión a DB, puede variar según el caso de uso o según el tipo de Base de Datos. Considerando esto y basándose en el principio de mantenibilidad de la aplicación, se puede definir el context en la capa handlers. Y pasarlo hasta el repository para que lo implemente. De esta forma si posteriormente se migra o se implemente otro repositorio, puede cambiarse el timeout según se requiera en el handler, sin tener que definir un context cada vez que se desarrolle un repository nuevo.

# IMPLEMENTANDO CONTEXT

STORAGE IMPLEMENTATION

# Cambios en la interfaz

Sobre la interfaz que se venía trabajando previamente, se añade un nuevo método llamado “GetOneWithContext”. El cual utilizará el context para la ejecución de la instrucción “Select”. Notar que este método también retorna un error.

```
type Repository interface {
    Store(models.Product) (models.Product, error)
    GetOne(id int) models.Product
    Update(product models.Product) (models.Product, error)
    GetAll() ([]models.Product, error)
    Delete(id int) error
    GetFullData(id int) models.Product
    GetOneWithcontext(ctx context.Context, id int) (models.Product, error)
}
```

# GetOneWithContext

```go
func (r *repository) GetOneWithcontext(ctx context.Context, id int) (models.Product, error) {
    var product models.Product
    db := db.StorageDB
    // se especifican estrictamente los campos necesarios en la query
    getQuery := "SELECT p.id, p.name, p.type, p.count, p.price FROM products p WHERE p.id = ?"
    // ya no se usa db.Query sino db.QueryContext
    rows, err := db.QueryContext(ctx, getQuery, id)
    if err != nil {
        log.Println(err)
        return product, err
    }
    for rows.Next() {
        if err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count, &product.Price); err != nil {
            log.Fatal(err)
            return product, err
        }
    }
    return product, nil
}
```

# GetOneWithContext

- Notar que en este método se implementó una de las recomendaciones previas, que consiste en establecer en la query, estrictamente los campos esperados de la consulta.

- Para este método se aplica el método db.QueryContext en vez de db.Query. Ya que es el método de la interfaz de database/sql que recibe como parámetro de entrada un context.

- Este nuevo método debe recibir un context ya previamente definido. A continuación se va a diseñar un test unitario en el que se creará un objeto nuevo del tipo context, y se comprobará el método GetOneWithContext.

# Testing GetOneWithContext

El método context.WithTimeout aplica el timeout a un context padre, el cual no existe en este caso. Por lo que se utiliza context.Background() que genera un context vacío.

{}

```go
func TestGetOneWithContext(t *testing.T) {
    // usamos un Id que exista en la DB
    id := 5
    // definimos un Product cuyo nombre sea igual al registro de la DB
    product := models.Product{
        Name: "test",
    }
    myRepo := NewRepo()
    // se define un context
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    productResult, err := myRepo.GetOneWithcontext(ctx, id)
    fmt.Println(err)
    assert.Equal(t, product.Name, productResult.Name)
}
```

# Testing GetOneWithContext

Finalmente procedemos a la ejecución del test:

```
$ go test
```

```
output
PASS
ok
meli-bootcamp-storage/internal/products 1.776s
```

# Testing GetOneWithContext

El test ejecutado anteriormente confirma que todo funciona correctamente el método, pero no pone a prueba el timeout del context. A efectos de esta comprobación, vamos a modificar el método GetOneWithContext

{}

```
func (r *repository) GetOneWithcontext(ctx context.Context, id int) (models.Product, error) {
    var product models.Product
    db := db.StorageDB
    // se utiliza una query que demore 30 segundos en ejecutarse
    getQuery := "SELECT SLEEP(30) FROM DUAL where 0 < ?"
    // ya no se usa db.Query sino db.QueryContext
    rows, err := db.QueryContext(ctx, getQuery, id)
    if err != nil {
        log.Println(err)
        return product, err
    }
    for rows.Next() {
        if err := rows.Scan(&product.ID, &product.Name, &product.Type, &product.Count,
&product.Price); err != nil {
            log.Fatal(err)
            return product, err
        }
    }
    return product, nil
}
```

# Testing GetOneWithContext

Ejecutamos nuevamente el test:

```
$ go test
```

```
output
PASS

2021/07/21 00:27:02 context deadline exceeded
context deadline exceeded
--- FAIL: TestGetOneWithContext (5.00s)
```

# Testing GetOneWithContext

El resultado del test, demuestra que se genera un error en la consulta, cuyo mensaje es “context deadline exceeded”. Esto demuestra la utilidad del context, ya que permite cancelar la consulta y devolver el error respectivo. Hay otros métodos que permiten manejar deadlines o cancelaciones. En general todos aplican no sólo para el manejo de DB sino a todos los flujos o procesos a los que se apliquen.

La documentación oficial de context puede encontrarse en:

https://pkg.go.dev/context

# RECURSOS

STORAGE IMPLEMENTATION

# Recursos

Para implementar los joins, es necesario aplicar cambios a la tabla products, además de la creación de la tabla warehouses

```
CREATE TABLE `warehouses` (
  `id` int(11) NOT NULL,
  `name` varchar(40) NOT NULL,
  `adress` varchar(150) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- Volcado de datos para la tabla `warehouses`
INSERT INTO `warehouses` (`id`, `name`, `adress`) VALUES
(1, 'Main Warehouse', '221b Baker Street');

ALTER TABLE `warehouses`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `warehouses`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=2;

ALTER TABLE `products` ADD `id_warehouse` INT NOT NULL AFTER `price`;

UPDATE `products` SET `id_warehouse` = '1';
```

# Gracias.

IT BOARDING

BOOTCAMP

Autor: Nelber Mora
Email: nelber.mora@digitalhouse.com
Última fecha de actualización: 21-07-21

IT BOARDING
BOOTCAMP