



Go-Lang GORM

Eko Kurniawan Khannedy

Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 12+ years experiences
- www.programmerzamannow.com
- youtube.com/c/ProgrammerZamanNow





Eko Kurniawan Khannedy

- Telegram : [@khannedy](https://t.me/khannedy)
- LinkedIn : <https://www.linkedin.com/company/programmer-zaman-now/>
- Facebook : fb.com/ProgrammerZamanNow
- Instagram : instagram.com/programmerzamannow
- Youtube : youtube.com/c/ProgrammerZamanNow
- Telegram Channel : t.me/ProgrammerZamanNow
- Tiktok : <https://tiktok.com/@programmerzamannow>
- Email : echo.khannedy@gmail.com



Sebelum Belajar

- Golang Dasar
- Golang Modules
- Golang Unit Test
- Golang Database

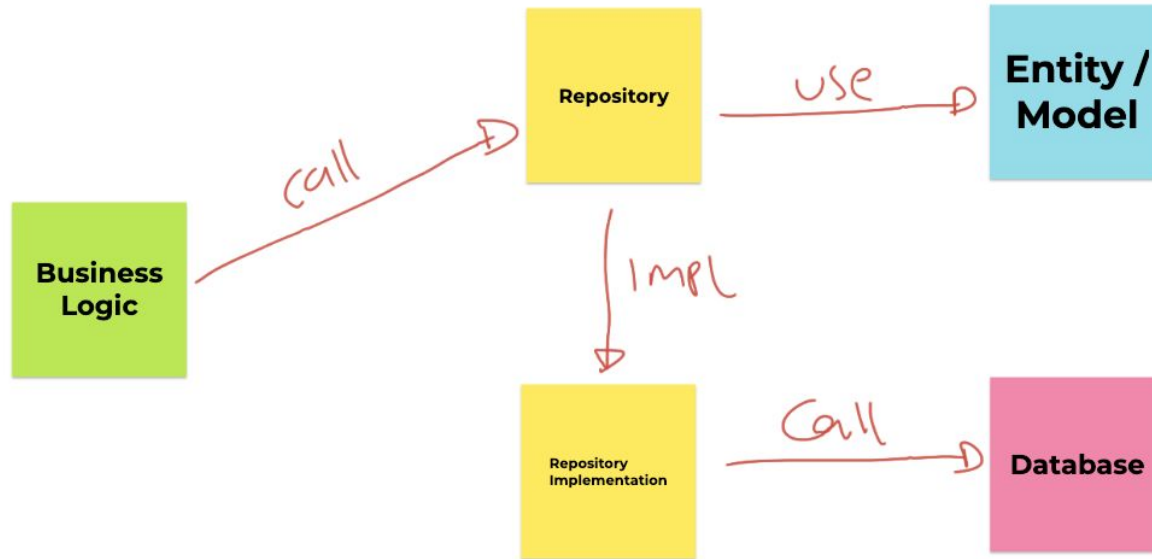
Pengenalan GORM



Object Relational Mapping

- ORM atau singkatan dari Object Relational Mapping, adalah teknik untuk memetakan data dari database relational ke dalam object dalam pemrograman
- Ketika kita belajar Golang Database, kita belajar pattern bernama Repository, yang digunakan sebagai jembatan komunikasi ke Database
- Saat membuat Repository, kita membuat struct Entity sebagai representasi dari Tabel di database
- Hal itu sebenarnya sudah bisa dibilang sebuah ORM, namun masih dilakukan secara manual

Diagram Repository Pattern





GORM

- GORM adalah salah satu library untuk implementasi ORM secara otomatis di Golang
- Dengan menggunakan GORM, kita bisa fokus membuat pemetaan struct Entity, tanpa harus memikirkan detail dari implementasi SQL yang harus kita buat untuk memanipulasi datanya
- GORM juga mendukung relasi antar Entity / Table, baik itu One to One, One to Many, sampai Many to Many
- <https://gorm.io/>

Membuat Project



Membuat Project

- Buat folder belajar-golang-gorm
- `go mod init belajar-golang-gorm`
- Tambahkan library Testify
- `go get github.com/stretchr/testify`



Menambah Library GORM

- `go get gorm.io/gorm`
- `go get gorm.io/driver/sqlite`
- `go get gorm.io/driver/mysql`
- `go get gorm.io/driver/postgres`
- `go get gorm.io/driver/sqlserver`
- `go get gorm.io/driver/clickhouse`

Setup Project



Membuat Database di MySQL

- `create database belajar_golang_gorm;`

Database Connection



Database Connection

- Hal pertama yang perlu kita lakukan sebelum menggunakan GORM, adalah membuat koneksi ke database
- Gunakan database yang sesuai dengan yang kita gunakan
- Untuk membuat koneksi ke database, kita bisa gunakan function `gorm.Open()`
- Tiap database memiliki config masing-masing, kita bisa lihat semua config di database di halaman ini :
- https://gorm.io/docs/connecting_to_the_database.html



Kode : Database Connection

```
import (  
    "gorm.io/driver/mysql"  
    "gorm.io/gorm"  
)  
  
func OpenConnection() *gorm.DB { 1 usage  new *  
    dialect := mysql.Open("root:@tcp(127.0.0.1:3306)/belajar_golang_gorm?charset=utf8mb4&parseTime=True&loc=Local")  
    db, err := gorm.Open(dialect, &gorm.Config{})  
    if err != nil {  
        panic(err)  
    }  
  
    return db  
}
```




Kode : Test Database Connection

```
var db = OpenConnection() // usage new *  
  
func TestOpenConnection(t *testing.T) { // new *  
    assert.NotNil(t, db)  
}
```

Raw SQL



Raw SQL

- Sebelum kita belajar dengan fitur ORM di GORM, kita akan bahas tentang bagaimana cara menggunakan Raw SQL
- Raw SQL artinya membuat query SQL secara manual
- Terdapat dua jenis SQL, untuk melakukan Query (Select), atau untuk mengubah data (Insert, Update, Delete)
- Untuk melakukan Query, kita bisa menggunakan method Raw(sql) di gorm.DB
- Dan untuk melakukan manipulasi data, bisa gunakan method Exec(sql) di gorm.DB



Kode : Table Sample

```
✓ create table sample
✓ (
    id    varchar(100) not null,
    name  varchar(100) not null,
    primary key (id)
) engine = InnoDB;
```



Kode : Execute SQL

```
func TestExecuteSQL(t *testing.T) {  
    err := db.Exec("insert into sample(id, name) values(?, ?)", "1", "Eko").Error  
    assert.Nil(t, err)  
  
    err = db.Exec("insert into sample(id, name) values(?, ?)", "2", "Budi").Error  
    assert.Nil(t, err)  
  
    err = db.Exec("insert into sample(id, name) values(?, ?)", "3", "Joko").Error  
    assert.Nil(t, err)  
  
    err = db.Exec("insert into sample(id, name) values(?, ?)", "4", "Rully").Error  
    assert.Nil(t, err)  
}
```

Kode : Query SQL

```
func TestRawSQL(t *testing.T) { new *
    var sample Sample
    err := db.Raw("select id, name from sample where id = ?", "1").Scan(&sample).Error
    assert.Nil(t, err)
    assert.Equal(t, "1", sample.ID)

    var samples []Sample
    err = db.Raw("select id, name from sample").Scan(&samples).Error
    assert.Nil(t, err)
    assert.Equal(t, 4, len(samples))
}
```



sql.Row & sql.Rows

- GORM sendiri sebenarnya didalamnya tetap menggunakan package sql bawaan dari Golang
- Jika kita ingin mendapatkan hasil Query dalam bentuk sql.Rows, kita bisa menggunakan method Rows() setelah melakukan Query



Kode : sql.Row

```
func TestSqlRow(t *testing.T) { new *  
    var samples []Sample  
  
    rows, err := db.Raw("select id, name from sample").Rows()  
    assert.Nil(t, err)  
    defer rows.Close()
```

```
    for rows.Next() {  
        var id string  
        var name string  
        err := rows.Scan(&id, &name)  
        assert.Nil(t, err)  
  
        samples = append(samples, Sample{  
            ID: id,  
            Name: name,  
        })  
    }  
  
    assert.Equal(t, 4, len(samples))
```




Kode : gorm.DB.ScanRows()

```
func TestScanRows(t *testing.T) { new *
    var samples []Sample

    rows, err := db.Raw("select id, name from sample").Rows()
    assert.Nil(t, err)
    defer rows.Close()

    for rows.Next() {
        err := db.ScanRows(rows, &samples)
        assert.Nil(t, err)
    }

    assert.Equal(t, 4, len(samples))
}
```

Model



Model

- Model atau Entity adalah struct representasi dari tabel di database
- Saat kita membuat tabel di database, direkomendasikan dibuatkan struct representasinya
- Hal ini agar kita tidak perlu melakukan pembuatan perintah SQL secara manual lagi



Kode : Tabel User

```
create table users
(
    id          varchar(100) not null,
    password    varchar(100) not null,
    name        varchar(100) not null,
    created_at  timestamp    not null default current_timestamp,
    updated_at  timestamp    not null default current_timestamp on update current_timestamp,
    primary key (id)
) engine = InnoDB;
```



Kode : User Entity

```
import "time"

type User struct { no usages new *
    ID      string
    Password string
    Name     string
    CreatedAt time.Time
    UpdatedAt time.Time
}
```



Convention

- Saat membuat struct, secara default GORM akan melakukan mapping secara otomatis, dimana nama tabel akan dipilih dari nama Struct menggunakan lower_case jamak, sedangkan nama kolom akan dipilih menggunakan lower_case.
- Selain itu, secara otomatis GORM akan memilih field ID sebagai primary key
- Namun, sebenarnya disarankan dibanding dilakukan secara otomatis menggunakan GORM, lebih baik kita deklarasikan secara manual menggunakan tag
- <https://gorm.io/docs/models.html#Fields-Tags>

Kode : User Entity dengan Tag

```
import "time"

type User struct { no usages  new *
    ID          string    `gorm:"primary_key;column:id"`
    Password     string    `gorm:"column:password"`
    Name         string    `gorm:"column:name"`
    CreatedAt    time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}
```



Table Name

- Secara default, nama tabel akan menggunakan lower_case dan jamak.
- Misal struct User akan menggunakan tabel users
- Misal struct OrderDetail akan menggunakan tabel order_details
- Namun jika kita ingin menggunakan manual nama tabel nya, kita bisa menggunakan interface `Tabler`, yang mewajibkan membuat method dengan nama `TableName()`



Kode : User Entity

```
type User struct { 1 usage  new *
    ID      string    `gorm:"primary_key;column:id"`
    Password string    `gorm:"column:password"`
    Name     string    `gorm:"column:name"`
    CreatedAt time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (u *User) TableName() string { 1 usage  new *
    return "users"
}
```

Model Conventions



Model Conventions

- Saat kita membuat struct untuk Model, jika kita mengikuti aturan dari GORM, sebenarnya kita tidak perlu menggunakan tag
- Sekarang kita akan bahas beberapa convention yang digunakan oleh GORM



ID Sebagai Primary Key

- GORM secara default menggunakan field ID sebagai primary key
- Jika kita membuat field ID di struct Model, secara default akan digunakan sebagai primary key di tabel
- Jika kita ingin memilih field lain selain ID, maka kita harus menggunakan tag gorm: "primaryKey"



Table Name

- GORM secara default akan menggunakan nama tabel dengan format snake_cases (lowercase, menggunakan _ sebagai pemisah, dan dijadikan jamak)
- Misal untuk struct User, secara default akan menggunakan nama tabel users
- Untuk struct OrderDetail secara default akan menggunakan nama tabel order_details
- Jika kita ingin mengubah nama default tabel nya, maka kita harus implementasi interface `Tabler` dengan menambahkan method `TableName()`



Column Name

- GORM secara default akan menggunakan snake_case untuk nama kolom dari struct Model yang kita buat
- Misal jika field ID artinya kolomnya id
- Jika field FirstName artinya kolomnya first_name
- Jika kita ingin mengubah nama kolom, kita bisa menggunakan tag gorm: "column:nama_kolom"



Timestamp Tracking

- GORM memiliki fitur timestamp tracking, yaitu melakukan perubahan otomatis untuk waktu dibuat dan diubah nya Model menggunakan field CreatedAt dan UpdatedAt
- CreatedAt dan UpdatedAt secara default akan menggunakan time.Now() ketika dibuat
- UpdatedAt akan selalu diubah menjadi time.Now() ketika diupdate datanya
- Jika kita ingin mengubah nama field nya, kita bisa tambahkan gorm:"autoCreateTime:true" untuk CreatedAt
- Dan menggunakan gorm:"autoUpdateTime:true" untuk UpdatedAt

Field Permission



Field Permission

- Secara default, semua field di struct Model akan dianggap kolom di tabel
- Dan semua perubahan di field, akan di update ke tabel di database
- Namun, kadang-kadang, mungkin kita ingin membuat field yang tidak merepresentasikan kolom di tabel sehingga tidak perlu di create / update
- Atau mungkin terdapat kolom yang tidak perlu di update lagi
- Untuk mendukung kasus seperti itu, GORM menyediakan Field Permission menggunakan tag gorm



GORM Field Permission

Tag	Keterangan
<-	Write permission, <-:create untuk create only, <-:update untuk update only, <- untuk create dan update
->	Read permission, ->:false untuk no read permission
-	ignore field ini, tidak ada read/write permission

Kode : User Entity

```
type User struct {
    ID          string    `gorm:"primary_key;column:id;<:-:create"`
    Password    string    `gorm:"column:password"`
    Name        Name      `gorm:"embedded"`
    CreatedAt   time.Time `gorm:"column:created_at;autoCreateTime;<:-:create"`
    UpdatedAt   time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    Information string    `gorm:"- "`
}

func (u *User) TableName() string {
    return "users"
}
```

Embedded Struct



Embedded Struct

- Secara default, field di Model akan menjadi kolom di tabel, kecuali yang menggunakan - field permission
- Namun bagaimana jika ternyata field yang kita buat sangat banyak?
- Kadang-kadang, ada baiknya kita simpan field yang sejenis dalam Struct yang berbeda
- Untungnya, GORM memiliki fitur bernama embedded struct, dimana kita bisa melakukan embed struct dalam field di Model, sehingga seluruh isi field di embedded struct akan dianggap field di Model utamanya
- Misal, kita akan tambahkan kolom first_name, middle_name dan last_name pada tabel users
- Lalu kita akan buat dalam embedded struct Name
- Kita bisa gunakan tag gorm: "embedded" untuk menandainya sebagai embedded struct



Kode : Alter Table users

```
✓ alter table users
    rename column name to first_name;

✓ alter table users
    add column middle_name varchar(100) null after first_name;

✓ alter table users
    add column last_name varchar(100) null after middle_name;
```



Kode : Struct Name

```
▼ type Name struct { 1 usage  new *  
    FirstName string `gorm:"column:first_name"`  
    MiddleName string `gorm:"column:middle_name"`  
    LastName string `gorm:"column:last_name"`  
}
```



Kode : User Model

```
type User struct {  
    ID          string    `gorm:"primary_key;column:id;<:-:create"`  
    Password    string    `gorm:"column:password"`  
    Name        Name      `gorm:"embedded"`  
    CreatedAt   time.Time `gorm:"column:created_at;autoCreateTime:<:-:create"`  
    UpdatedAt   time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`  
    Information string    `gorm:"-`  
}  
  
func (u *User) TableName() string {  
    return "users"  
}
```

Create



Create

- Untuk memasukkan data ke database, kita tidak perlu membuat SQL Insert secara manual
- GORM bisa membaca data dari Model yang sudah kita buat, lalu secara otomatis akan membuatkan perintah SQL sesuai dengan data Model yang kita buat
- Kita bisa menggunakan method `Create()` pada `gorm.DB`



Kode : Create

```
✓ func TestCreateUser(t *testing.T) { new *
✓     user := User{
        ID:      "1",
        Password: "rahasia",
        Name: Name{
            FirstName: "Eko",
            MiddleName: "Kurniawan",
            LastName:  "Khannedy",
        },
        Information: "Ini akan di ignore",
    }
}
```

```
response := db.Create(user)
assert.Nil(t, response.Error)
assert.Equal(t, 1, int(response.RowsAffected))
```



Batch Insert

- Saat kita menggunakan `Create()` Method, maka tiap data akan dibuatkan SQL Insert
- Kadang ketika kita ingin memasukkan banyak data sekaligus, ada baiknya kita menggunakan sekali SQL Insert agar lebih efektif
- GORM mendukung hal ini menggunakan method `Create(slices)`, atau jika ingin menentukan jumlah data per SQL Insert, kita bisa gunakan `CreateInBatches(slices, size)`



Kode : Batch Insert

```
func TestBatchInsert(t *testing.T) {  
    var users []User  
    for i := 2; i < 10; i++ {  
        users = append(users, User{  
            ID: strconv.Itoa(i),  
            Name: Name{  
                FirstName: "User " +  
                    strconv.Itoa(i),  
            },  
            Password: "rahasia",  
        })  
    }  
}
```

```
result := db.Create(&users)  
assert.Nil(t, result.Error)  
assert.Equal(t, 8, int(result.RowsAffected))
```

—

Logger



Logger

- Secara default, informasi perintah SQL yang dieksekusi oleh GORM tidak akan di log
- Kita bisa mengubah level log dari GORM menggunakan `gorm.Config{}` ketika membuat koneksi database



Kode : Mengubah Logger

```
func OpenConnection() *gorm.DB { 1 usage new *
    dialect := mysql.Open("root:@tcp(127.0.0.1:3306)/belajar_golang_gorm?charset=utf8mb4&parseTime=True&loc=Local")
    db, err := gorm.Open(dialect, &gorm.Config{
        Logger: logger.Default.LogMode(logger.Info),
    })
    if err != nil {
        panic(err)
    }

    return db
}
```

Transaction



Transaction

- Transaction hanya bisa terjadi jika kita menggunakan Database Connection yang sama
- Saat kita menggunakan Method di gorm.DB, bisa saja tiap Method akan menggunakan Database Connection yang berbeda, karena Connection Pool nya diatur oleh GORM
- Jika kita ingin melakukan transaction, kita bisa menggunakan method Transaction(callback), dan di dalam function callback kita bisa buat semua kode transaction nya



Kode : Transaction Success

```
func TestTransaction(t *testing.T) { new *
    err := db.Transaction(func(tx *gorm.DB) error {
        err := tx.Create(&User{ID: "10", Password: "rahasia", Name: Name{FirstName: "User 10"}}).Error
        if err != nil {
            return err
        }

        err = tx.Create(&User{ID: "11", Password: "rahasia", Name: Name{FirstName: "User 11"}}).Error
        if err != nil {
            return err
        }

        err = tx.Create(&User{ID: "12", Password: "rahasia", Name: Name{FirstName: "User 12"}}).Error
        if err != nil {
            return err
        }
        return nil
    })

    assert.Nil(t, err)
```



Kode : Transaction Error

```
func TestTransactionError(t *testing.T) { new *
    err := db.Transaction(func(tx *gorm.DB) error {
        err := tx.Create(&User{ID: "13", Password: "rahasia", Name: Name{FirstName: "User 10"}}).Error
        if err != nil {
            return err
        }

        err = tx.Create(&User{ID: "11", Password: "rahasia", Name: Name{FirstName: "User 11"}}).Error
        if err != nil {
            return err
        }
        return nil
    })

    assert.NotNil(t, err)
}
```



Manual Transaction

- Selain menggunakan Method Transaction(callback) , kita juga bisa melakukan manajemen transaksi secara manual
- Kita bisa membuat object gorm.DB baru ketika menjalankan transaksi menggunakan method Begin()
- Ketika ingin melakukan commit, kita bisa menggunakan method Commit()
- Ketika ingin melakukan rollback, kita bisa menggunakan method Rollback()



Kode : Manual Transaction Success

```
func TestManualTransactionSuccess(t *testing.T) { new *
    tx := db.Begin()
    defer tx.Rollback()

    err := tx.Create(&User{ID: "13", Password: "rahasia", Name: Name{FirstName: "User 13"}}).Error
    assert.Nil(t, err)

    err = tx.Create(&User{ID: "14", Password: "rahasia", Name: Name{FirstName: "User 14"}}).Error
    assert.Nil(t, err)

    if err == nil {
        tx.Commit()
    }
}
```



Kode : Manual Transaction Error

```
func TestManualTransactionError(t *testing.T) { new *
    tx := db.Begin()
    defer tx.Rollback()

    err := tx.Create(&User{ID: "15", Password: "rahasia", Name: Name{FirstName: "User 15"}}).Error
    assert.Nil(t, err)

    err = tx.Create(&User{ID: "14", Password: "rahasia", Name: Name{FirstName: "User 14"}}).Error
    assert.NotNil(t, err)

    if err == nil {
        tx.Commit()
    }
}
```

Query



Query

- Sama dengan Create(), GORM juga bisa secara otomatis membuat SQL SELECT dari Model yang kita buat
- Sehingga kita tidak perlu lagi membuat SQL SELECT secara manual



Single Object

- GORM menyediakan beberapa Method untuk mengambil single object dari database
- First(), digunakan untuk mengambil data pertama yang diurutkan berdasarkan id
- Take(), digunakan untuk mengambil satu data, tanpa diurutkan
- Last(), digunakan untuk mengambil data terakhir yang diurutkan berdasarkan id
- Jika data tidak ditemukan, maka akan error `gorm.ErrRecordNotFound`



Kode : Single Object

```
func TestQuerySingleObject(t *testing.T) { new *
    user := User{}
    result := db.First(&user)
    assert.Nil(t, result.Error)
    assert.Equal(t, "1", user.ID)

    user = User{}
    result = db.Last(&user)
    assert.Nil(t, result.Error)
    assert.Equal(t, "9", user.ID)
}
```



Inline Condition

- Saat kita menggunakan method First, Take atau Last, terdapat parameter selanjutnya bernama Inline Condition
- Inline Condition tersebut secara otomatis akan menjadi kondisi WHERE di SQL SELECT nya



Kode : Inline Condition

```
▼ func TestQueryInlineCondition(t *testing.T) { new *  
    user := User{}  
    result := db.First(&user, "id = ?", "5")  
    assert.Nil(t, result.Error)  
    assert.Equal(t, "5", user.ID)  
}
```



Query All Objects

- GORM juga bisa digunakan untuk melakukan QUERY untuk semua data di tabel menggunakan method Find()
- Sama seperti method sebelumnya, method Find() juga mendukung Inline Condition jika kita mau tambahkan kondisi WHERE nya
- Untuk detail query WHERE, akan kita bahas nanti di materi Query Condition



Kode : Query All Objects

```
func TestQueryAllObjects(t *testing.T) { new *
    var users []User
    result := db.Find(&users, "id in ?", []string{"1", "2", "3", "4"})
    assert.Nil(t, result.Error)
    assert.Equal(t, 4, len(users))
}
```

Advanced Query



Query Condition

- Sebelumnya kita sudah bisa menggunakan Inline Parameter ketika melakukan Query
- Selain menggunakan Inline Parameter, kita juga bisa menggunakan method Where() untuk mengubah kondisi query yang akan kita buat



Kode : Query Condition

```
func TestQueryCondition(t *testing.T) { new *
    var users []User
    result := db.Where("first_name like ?", "%User%").
        Where("password = ?", "rahasia").
        Find(&users)
    assert.Nil(t, result.Error)

    assert.Equal(t, 13, len(users))
}
```



Or Condition

- Secara default saat kita menggunakan Where(), kondisi akan digabung menggunakan AND operator
- Jika kita ingin menggunakan OR operator, kita bisa menggunakan method Or()



Kode : OR Operator

```
func TestOrOperator(t *testing.T) { new *
    var users []User
    result := db.Where("first_name like ?", "%User%").
        Or("password = ?", "rahasia").
        Find(&users)
    assert.Nil(t, result.Error)

    assert.Equal(t, 14, len(users))
}
```



Not Condition

- Dan jika kita ingin menggunakan NOT Operator, kita bisa menggunakan method Not()



Kode : Not Operator

```
▼ func TestNotOperator(t *testing.T) { new *
    var users []User
    result := db.Not("first_name like ?", "%User%").
        Where("password = ?", "rahasia").
        Find(&users)
    assert.Nil(t, result.Error)

    assert.Equal(t, 1, len(users))
}
```



Select Fields

- Secara default, semua kolom akan di select dan dimasukkan ke field Model
- Jika misal kita ingin menentukan kolom apa saja yang mau di select, kita bisa menggunakan method `Select(columns)`




Kode : Select Fields

```
func TestSelectFields(t *testing.T) { new *
    var users []User
    result := db.Select("id", "first_name").Find(&users)
    assert.Nil(t, result.Error)

    for _, user := range users {
        assert.NotNil(t, user.ID)
        assert.Equal(t, "", user.Name.FirstName)
    }

    assert.Equal(t, 14, len(users))
}
```





Struct & Map Condition

- Saat kita menggunakan `Where()`, `Not()` dan `Or()`, kita juga bisa menggunakan parameter Struct atau Map
- Secara otomatis field atau key akan dijadikan kolom query, dan value akan dijadikan value query
- Ini cocok ketika pada kasus kita butuh query yang dinamis, sehingga kolom yang dicari bisa berbeda-beda sesuai kondisi pencarian



Kode : Struct Condition

```
func TestStructCondition(t *testing.T) { new *
    userCondition := User{
        Name: Name{
            FirstName: "User 5",
        },
    }
    var users []User
    result := db.Where(userCondition).Find(&users)
    assert.Nil(t, result.Error)
    assert.Equal(t, 1, len(users))
}
```



Kode : Map Condition

```
func TestMapCondition(t *testing.T) { new *
    mapCondition := map[string]interface{}{
        "middle_name": "",
    }
    var users []User
    result := db.Where(mapCondition).Find(&users)
    assert.Nil(t, result.Error)
    assert.Equal(t, 13, len(users))
}
```



Order, Limit dan Offset

- Untuk melakukan sorting, kita juga bisa menggunakan method `Order()`
- Dan untuk melakukan paging, kita bisa menggunakan method `Limit()` dan `Offset()`



Kode : Order, Limit dan Offset

```
✓ func TestOrderLimitOffset(t *testing.T) { new *
    var users []User
    result := db.Order("id asc, first_name asc").Limit(5).Offset(5).Find(&users)
    assert.Nil(t, result.Error)
    assert.Equal(t, 5, len(users))
    assert.Equal(t, "14", users[0].ID)
}
```

Query Non Model



Query Non Model

- Saat kita menggunakan `First()`, `Take()`, `Last()` dan `Find()`, GORM melihat struktur tabel ke Model yang kita gunakan
- Namun GORM memiliki fitur untuk menyimpan data ke data yang bukan Model, contoh, kita hanya ingin melakukan query `first_name` dan `last_name` saja di tabel `users` misalnya. Kita bisa membuat struct berbeda dibanding menggunakan model `User`
- Jika kita ingin melakukan hal ini, GORM tetap harus tahu, Model mana yang digunakan, caranya adalah kita bisa menggunakan method `Model()` untuk menentukan Model yang digunakan



Kode : Query Non Model

```
type UserResponse struct {
    ID          string
    FirstName   string
    LastName    string
}

func TestQueryNonModel(t *testing.T) {
    var users []UserResponse
    result := db.Model(&User{}).Select("id", "first_name", "last_name").Find(&users)
    assert.Nil(t, result.Error)
    assert.Equal(t, 14, len(users))
}
```

Update



Update

- Untuk melakukan update data Model yang sudah kita modifikasi, kita bisa menggunakan method `Save()` di `gorm.DB`
- Secara otomatis semua kolom yang memang memiliki permission untuk di update, akan di update ke database



Kode : Update

```
func TestUpdate(t *testing.T) { new *
    user := User{}
    result := db.First(&user, "id = ?", "1")
    assert.Nil(t, result.Error)

    user.Name.FirstName = "Budi"
    user.Name.MiddleName = ""
    user.Name.LastName = "Nugraha"
    user.Password = "password123"
    result = db.Save(&user)
    assert.Nil(t, result.Error)
}
```



Update Selected Column

- Secara default, melakukan `Save()` untuk data Model, akan melakukan update semua kolom, walaupun tidak berubah
- Jika kita ingin menentukan hanya beberapa kolom yang ingin di update, kita bisa menggunakan method `Updates()`
- Atau menggunakan `Update(kolom, value)` jika hanya ingin melakukan update satu kolom saja



Kode : Update Selected Columns

```
func TestSelectedColumns(t *testing.T) { new *
    result := db.Model(&User{}).Where("id = ?", "1").Updates(map[string]interface{}{
        "middle_name": "",
        "last_name":   "Morro",
    })
    assert.Nil(t, result.Error)

    result = db.Model(&User{}).Where("id = ?", "1").Update("password", "diubahlagi")
    assert.Nil(t, result.Error)

    result = db.Where("id = ?", "1").Updates(User{
        Name: Name{
            FirstName: "Eko",
            LastName:  "Khannedy",
        },
    })
    assert.Nil(t, result.Error)
}
```

Auto Increment



Auto Increment

- Salah satu fitur yang biasa ada di database adalah Auto Increment untuk Primary Key
- Contoh di MySQL ada AUTO_INCREMENT, atau di PostgreSQL ada SERIAL
- GORM mendukung pembuatan ID yang Auto Increment, dan secara otomatis akan melakukan query data ID setelah kita Create() datanya, sehingga tidak perlu melakukan query manual lagi
- Untuk memberitahu bahwa field adalah auto increment, kita harus gunakan tag `gorm:"autoIncrement"`



Kode : Tabel user_logs

```
✓ create table user_logs
✓ (
    id          int auto_increment,
    user_id     varchar(100) not null,
    action      varchar(100) not null,
    created_at  timestamp     not null default current_timestamp,
    updated_at  timestamp     not null default current_timestamp on update current_timestamp,
    primary key (id)
) engine = InnoDB;
```




Kode : UserLog Model

```
import "time"

type UserLog struct { 2 usages new *
    ID          int          `gorm:"primary_key;column:id;autoIncrement"`
    UserID       string         `gorm:"column:user_id"`
    Action       string         `gorm:"column:action"`
    CreatedAt    time.Time      `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time      `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (u *UserLog) TableName() string { 1 usage new *
    return "user_logs"
}
```



Kode : Insert User Log

```
> func TestAutoIncrement(t *testing.T) { new *
  >   for i := 0; i < 10; i++ {
  >     userLog := UserLog{
  >       UserID: "1",
  >       Action: "Test Action",
  >     }
  >     result := db.Create(&userLog)
  >     assert.Nil(t, result.Error)
  >     assert.NotEqual(t, 0, userLog.ID)
  >     fmt.Println(userLog.ID)
  >   }
  > }
```

Timestamp Tracking



Timestamp Tracking

- Seperti yang sudah kita bahas di materi Conventions, bahwa GORM menggunakan field CreatedAt dan UpdatedAt sebagai Timestamp Tracking
- Atau jika ingin menggunakan field yang berbeda, kita bisa menambahkan tag gorm:"autoCreateTime" atau gorm:"autoUpdateTime"



Tipe Data Timestamp Tracking

- Sebelumnya kita menggunakan tipe data `time.Time` sebagai field untuk Timestamp Tracking
- Namun, sebenarnya kita juga bisa ubah tipe datanya jika kita mau
- GORM mendukung tipe data dalam bentuk number, dimana satuannya bisa kita ganti menjadi milli untuk millisecond, atau nano untuk nanosecond, semuanya disimpan dalam waktu epoch unix time
- Misal gorm: "autoCreateTime:milli" atau gorm: "autoUpdateTime:milli"
- <https://currentmillis.com/>



Kode : Alter Table User Log

```
delete from user_logs;
```

```
✓ alter table user_logs  
    modify created_at bigint not null;
```

```
✓ alter table user_logs  
    modify updated_at bigint not null;
```

Kode : UserLog Model

```
type UserLog struct { 2 usages new *
    ID      int      `gorm:"primary_key;column:id;autoIncrement"`
    UserID   string   `gorm:"column:user_id"`
    Action   string   `gorm:"column:action"`
    CreatedAt int64    `gorm:"column:created_at;autoCreateTime:milli"`
    UpdatedAt int64    `gorm:"column:updated_at;autoCreateTime:milli;autoUpdateTime:milli"`
}

func (u *UserLog) TableName() string { 1 usage new *
    return "user_logs"
}
```



Kode : Insert User Log

```
> func TestAutoIncrement(t *testing.T) { new *
  >   for i := 0; i < 10; i++ {
  >     userLog := UserLog{
  >       UserID: "1",
  >       Action: "Test Action",
  >     }
  >     result := db.Create(&userLog)
  >     assert.Nil(t, result.Error)
  >     assert.NotEqual(t, 0, userLog.ID)
  >     fmt.Println(userLog.ID)
  >   }
  > }
```

Upsert



Save

- Sebelumnya, kita telah menggunakan method Save() untuk melakukan UPDATE
- Method Save() sebenarnya memiliki kemampuan untuk mendeteksi apakah harus melakukan Update atau Create
- Jika data yang kita kirim tidak memiliki value ID, maka secara default akan melakukan Create
- Jika data yang kita kirim memiliki value ID, maka akan melakukan Update
- Hal ini mungkin cocok untuk jenis data yang ID nya adalah Auto Increment, karena kita tidak butuh ID ketika melakukan Create



Kode : Save

```
func TestSaveOrUpdate(t *testing.T) { new *
    userLog := UserLog{
        UserID: "1",
        Action: "Test Action",
    }
    result := db.Save(&userLog) // create
    assert.Nil(t, result.Error)

    userLog.UserID = "2"
    result = db.Save(&userLog) // update
    assert.Nil(t, result.Error)
}
```



Data Non Auto Increment

- Bagaimana dengan jenis data yang memiliki ID tidak auto increment, misal data User sebelumnya
- Untungnya, `Save()` juga bisa digunakan untuk proses otomatis Create
- Jadi `Save()` akan mencoba melakukan Update terlebih dahulu, ketika mendeteksi jumlah `Effecte`dRow nya adalah 0, secara otomatis `Save()` akan melakukan proses Create

Kode : Save Non Auto Increment Data

```
func TestSaveOrUpdateNonAutoIncrement(t *testing.T) { new *
    user := User{
        ID: "99",
        Name: Name{
            FirstName: "User 99",
        },
    }
    result := db.Save(&user) // create
    assert.Nil(t, result.Error)

    user.Name.FirstName = "User 99 Updated"
    result = db.Save(&user) // create
    assert.Nil(t, result.Error)
}
```



On Conflict

- GORM juga menawarkan pengaturan Conflict di method Create()
- Dengan pengaturan ini, kita bisa menentukan ketika kita coba Create() data, lalu terjadi conflict (data sudah ada), apa yang mau kita lakukan?
- Kita bisa mengubah pengaturan conflict ini menggunakan method Clauses()



Kode : On Conflict

```
func TestConflict(t *testing.T) { new *
    user := User{
        ID: "88",
        Name: Name{
            FirstName: "User 88",
        },
    }
    result := db.Clauses(clause.OnConflict{
        UpdateAll: true,
    }).Save(&user) // create
    assert.Nil(t, result.Error)
}
```

—

Delete



Delete

- Untuk melakukan penghapusan data, kita bisa menggunakan method Delete()
- Method Delete() juga mendukung Inline Condition atau menggunakan Where() method



Kode : Delete

```
func TestDelete(t *testing.T) { new *
    var user User
    result := db.First(&user, "id = ?", "88")
    assert.Nil(t, result.Error)
    result = db.Delete(&user)
    assert.Nil(t, result.Error)

    result = db.Delete(&User{}, "id = ?", "99")
    assert.Nil(t, result.Error)

    result = db.Where("id = ?", "77").Delete(&User{})
    assert.Nil(t, result.Error)
}
```



Soft Delete



Soft Delete

- Soft Delete adalah salah satu praktek yang sering dilakukan ketika membuat aplikasi
- Soft Delete merupakan praktek menghapus data, tanpa menghapus data dari database
- Praktek ini membuat satu kolom biasanya berupa tipe data timestamp yang berisi waktu dihapus
- Jika kolom tersebut berisi data, otomatis data tersebut dianggap sudah di delete



GORM Soft Delete

- GORM Mendukung fitur Soft Delete secara otomatis, caranya kita cukup membuat field DeletedAt dengan time gorm.DeletedAt (alias untuk time.Time)
- Jika GORM mendeteksi terdapat field dengan nama DeletedAt, secara otomatis GORM akan melakukan Soft Delete
- Selain itu, ketika melakukan Query, secara otomatis juga GORM akan menambah filter Soft Delete, yang artinya hasil query hanya data yang belum di delete



Kode : Table Todo

```
✓ create table todos
✓ (
    id            bigint      not null auto_increment,
    user_id       varchar(100) not null,
    title         varchar(100) not null,
    description    text        null,
    created_at    timestamp    not null default current_timestamp,
    updated_at    timestamp    not null default current_timestamp on update current_timestamp,
    deleted_at    timestamp    null,
    primary key (id)
) engine = InnoDB;
```

Kode : Todo Model

```
type Todo struct { 1 usage new *
    ID          int64          `gorm:"primary_key;column:id;autoIncrement"`
    UserID       string           `gorm:"column:user_id"`
    Title        string           `gorm:"column:title"`
    Description   string           `gorm:"column:description"`
    CreatedAt     time.Time        `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt     time.Time        `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    DeletedAt     gorm.DeletedAt  `gorm:"column:deleted_at"`
}


func (t *Todo) TableName() string { 1 usage new *
    return "todos"
}
```



Kode : Delete Todo

```
func TestSoftDelete(t *testing.T) {  
    todo := Todo{  
        UserID:      "1",  
        Title:       "Todo 1",  
        Description: "Isi Todo 1",  
    }  
    result := db.Create(&todo)  
    assert.Nil(t, result.Error)
```

```
    result = db.Delete(&todo)  
    assert.Nil(t, result.Error)  
    assert.NotNil(t, todo.DeletedAt)  
  
    var todos []Todo  
    result = db.Find(&todos)  
    assert.Nil(t, result.Error)  
    assert.Equal(t, 0, len(todos))
```





Unscoped

- Kita kita ingin mengambil data termasuk yang sudah di soft delete, kita bisa gunakan method `Unscoped()`
- Method `Unscoped()` juga bisa digunakan jika kita benar-benar mau melakukan hard delete permanen di database



Kode : Unscoped

```
func TestUnscoped(t *testing.T) { new *
    var todo Todo
    result := db.Unscoped().First(&todo, "id = ?", "1")
    assert.Nil(t, result.Error)

    result = db.Unscoped().Delete(&todo)
    assert.Nil(t, result.Error)

    var todos []Todo
    result = db.Unscoped().Find(&todos)
    assert.Nil(t, result.Error)
    assert.Equal(t, 0, len(todos))
}
```



Peringatan

- Saat menggunakan Soft Delete, perhatikan penggunaan Primary Key atau Unique Index
- Ketika data sudah dihapus secara soft delete, sebenarnya data masih ada di tabel, oleh karena itu pastikan data primary key atau unique index tidak duplicate dengan data yang sudah dihapus secara soft delete

Model Struct



Model Struct

- GORM menyediakan sebuah struct bernama Model yang berisi field ID, CreatedAt, UpdatedAt dan DeletedAt
- Ini cocok digunakan ketika kita menggunakan field yang sesuai dengan convention nya GORM
- Contoh misal kita bisa gunakan struct Model ini ketika membuat model Todo



Kode : Struct gorm.Model

```
4
5  ✓ // Model a basic GoLang struct which includes the following fields: ID, CreatedAt, UpdatedAt, DeletedAt
6    // It may be embedded into your model or you may build your own model without it
7    //
8    // type User struct {
9    //     gorm.Model
10   // }
11  ✓ type Model struct {
12      ID          uint `gorm:"primaryKey"`
13      CreatedAt   time.Time
14      UpdatedAt   time.Time
15      DeletedAt   DeletedAt `gorm:"index"`
16  }
17
```



Kode : Todo Model

```
import "gorm.io/gorm"

type UserLog struct { 3 usages new *
    gorm.Model
    UserID string `gorm:"column:user_id"`
    Action string `gorm:"column:action"`
}

func (u *UserLog) TableName() string { 1 usage new *
    return "user_logs"
}
```

—

Lock



Lock

- Hal yang biasa kita lakukan saat menggunakan database adalah melakukan Lock data
- Biasanya, ini dilakukan agar tidak terjadi RACE CONDITION ketika memanipulasi data yang sama oleh beberapa request
- Untuk melakukan Lock menggunakan GORM, kita bisa menambahkan Clauses() Locking
- Kita bisa menentukan jenis Lock nya, apakah itu UPDATE, SHARE atau yang lainnya. Sesuai dengan dukungan database yang kita gunakan

Kode : Lock

```
func TestLock(t *testing.T) { new *
    err := db.Transaction(func(tx *gorm.DB) error {
        var user User
        err := tx.Clauses(clause.Locking{Strength: "UPDATE"}).First(&user, "id = ?", "1").Error
        if err != nil {
            return err
        }

        user.Name.FirstName = "Joko"
        user.Name.LastName = "Morro"
        return tx.Save(&user).Error
    })

    assert.Nil(t, err)
}
```

One to One



One to One

- Relationship dalam database yang paling sederhana adalah One to One, dimana data di tabel berelasi dengan satu data di tabel lain
- Di GORM, One to One disebut juga relasi Has One
- Untuk membuat relasi One to One di Model sangat mudah, kita cukup buat field dengan tipe Model yang berelasi
- Lalu kita bisa tambahkan informasi di tag
- `gorm:"foreignKey:nama_kolom"` untuk kolom yang dijadikan foreign key
- `gorm:"references:nama_kolom"` untuk kolom yang dijadikan reference



Kode : Tabel Wallets

```
✓ create table wallets
✓ (
    id          varchar(100) not null,
    user_id     varchar(100) not null,
    balance     bigint       not null,
    created_at  timestamp    not null default current_timestamp,
    updated_at  timestamp    not null default current_timestamp on update current_timestamp,
    primary key (id),
    foreign key (user_id) references users (id)
) engine = InnoDB;
```



Kode : Wallet Model

```
import "time"

type Wallet struct { 1usage new *
    ID          string    `gorm:"primary_key;column:id"`
    UserId       string    `gorm:"column:user_id"`
    Balance      int64     `gorm:"column:balance"`
    CreatedAt    time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (w *Wallet) TableName() string { new *
    return "wallets"
}
```

Kode : User Model

```
↑ 36 usages new *
type User struct {
    ID          string    `gorm:"primary_key;column:id;<:-:create"`
    Password    string    `gorm:"column:password"`
    Name        Name      `gorm:"embedded"`
    CreatedAt   time.Time `gorm:"column:created_at;autoCreateTime;<:-:create"`
    UpdatedAt   time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    Information string    `gorm:"- "`
    Wallet      Wallet    `gorm:"foreignKey:user_id;references:id"`
}
```



Kode : Create Wallet

```
func TestCreateWallet(t *testing.T) { new *
    wallet := Wallet{
        ID:      "1",
        UserId:  "1",
        Balance: 1000000,
    }
    err := db.Create(&wallet).Error
    assert.Nil(t, err)
}
```




Preload

- Secara default, relasi tidak akan di query oleh GORM, artinya sifatnya adalah LAZY
- Jika kita ingin melakukan query relation (EAGER) secara langsung ketika melakukan query Model, kita bisa sebutkan relasi yang ingin kita load menggunakan method Preload()



Kode : Retrieve Relation (1)

```
func TestRetrieveRelation(t *testing.T) { new *
    var user User
    err := db.Model(&User{}).Preload("Wallet").First(&user).Error
    assert.Nil(t, err)

    assert.Equal(t, "1", user.ID)
    assert.Equal(t, "1", user.Wallet.ID)
}
```



Join

- Menggunakan Preload(), GORM akan melakukan pengambilan data relasi menggunakan query yang terpisah
- Hal ini cocok ketika menggunakan relasi One to Many atau Many to Many
- Namun pada kasus One to One, kadang ada baiknya kita lakukan sekali query saja menggunakan JOIN, karena hasilnya hanya satu data, jadi lebih cepat
- Jika kita ingin menggunakan Join, kita bisa menggunakan method Joins(), lalu menyebutkan field mana yang akan kita lakukan JOIN



Kode : Join

```
✓ func TestRetrieveRelationJoin(t *testing.T) { new *
    var users []User
    err := db.Model(&User{}).Joins("Wallet").Find(&users).Error
    assert.Nil(t, err)

    assert.Equal(t, 14, len(users))
}
```

Auto Upsert Relation



Auto Create/Update

- Saat kita menggunakan relasi, lalu kita ingin melakukan create/update data Model, secara default GORM akan mengecek relasi yang terdapat di data tersebut
- Jika terdapat data relasi, GORM akan melakukan proses Upsert terhadap data relasinya, sehingga kita tidak perlu melakukan create/update data relasi secara manual



Kode : Auto Create/Update

```
func TestAutoCreateUpdate(t *testing.T) { new *
    user := User{
        ID:      "20",
        Password: "rahasia",
        Name: Name{
            FirstName: "User 20",
        },
        Wallet: Wallet{
            ID:      "20",
            UserId:  "20",
            Balance: 1000000,
        },
    }
    err := db.Create(&user).Error
    assert.Nil(t, err)
}
```



Skip Auto Create/Update

- Jika kita tidak mau melakukan auto create / update data relasi, kita bisa gunakan method `Omit()` yang berisi `clause.Associations`
- Ini memberitahu GORM bahwa kita tidak mau melakukan auto create/update untuk data relasinya



Kode : Skip Auto Create/Update

```
func TestSkipAutoCreateUpdate(t *testing.T) { new *
    user := User{
        ID:      "21",
        Password: "rahasia",
        Name: Name{
            FirstName: "User 21",
        },
        Wallet: Wallet{
            ID:      "21",
            UserId:  "21",
            Balance: 1000000,
        },
    }
    err := db.Omit(cclause.Associations).Create(&user).Error
    assert.Nil(t, err)
}
```

One to Many



One to Many

- Relasi One to Many adalah relasi dimana data di tabel bisa memiliki relasi ke banyak data di tabel lain
- Di GORM, One to Many juga disebut relasi Has Many
- Untuk membuat relasi One to Many, kita bisa gunakan field dengan tipe Slice Model yang berelasi
- Kita juga bisa menentukan informasi seperti foreignKey dan references nya, sama seperti ketika menggunakan relasi One to One



Kode : Table Addresses

```
create table addresses
(
  id          bigint          not null auto_increment,
  user_id     varchar(100)    not null,
  address     varchar(100)    not null,
  created_at  timestamp       not null default current_timestamp,
  updated_at  timestamp       not null default current_timestamp on update current_timestamp,
  primary key (id),
  foreign key (user_id) references users (id)
) engine = InnoDB;
```

Kode : Address Model

```
type Address struct { 1 usage new *
    ID          int64      `gorm:"primary_key;column:id;autoIncrement"`
    UserId       string     `gorm:"column:user_id"`
    Address      string     `gorm:"column:address"`
    CreatedAt    time.Time  `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time  `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (a *Address) TableName() string { new *
    return "addresses"
} =
```



Kode : User Model

```
✓ type User struct { 42 usages new *
    ID          string    `gorm:"primary_key;column:id;< -:create"`
    Password     string    `gorm:"column:password"`
    Name         Name      `gorm:"embedded"`
    CreatedAt    time.Time `gorm:"column:created_at;autoCreateTime;< -:create"`
    UpdatedAt    time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    Information  string    `gorm:"- "`
    Wallet       Wallet   `gorm:"foreignKey:user_id;references:id"`
    Addresses    []Address `gorm:"foreignKey:user_id;references:id"`
}
```

Kode : Auto Create/Update

```
func TestUserAndAddresses(t *testing.T) { new
    user := User{
        ID:      "50",
        Password: "rahasia",
        Name:     Name{FirstName: "User 50"},
        Wallet:   Wallet{
            ID:      "50",
            UserId:  "50",
            Balance: 1000000,
        },
        Addresses: []Address{
```

```
        Addresses: []Address{
            {
                UserId:  "50",
                Address: "Jalan A",
            },
            {
                UserId:  "50",
                Address: "Jalan B",
            },
        },
    }

    err := db.Create(&user).Error
    assert.Nil(t, err)
```



Kode : Preload & Join

```
func TestPreloadJoinOneToMany(t *testing.T) { new *
    var usersPreload []User
    err := db.Model(&User{}).Preload("Addresses").Joins("Wallet").Find(&usersPreload).Error
    assert.Nil(t, err)
}
```

Belongs To



Belongs To di One to Many

- Saat kita membuat relasi One to Many, ada sudut pandang lain dari Model sebelumnya, yaitu relasi Many to One
- Pada kasus ini, kita bisa menggunakan relasi Belongs To (milik) di GORM
- Contoh sebelumnya kita tahu bahwa User punya banyak Address, artinya Address milik (belongs to) User
- Kita bisa tambahkan relasi ini di model Address, agar ketika kita melakukan query ke model Address, kita juga bisa mendapatkan informasi relasi User nya
- Cara membuatnya mirip seperti ketika membuat relasi One to One



Kode : Address Model

```
type Address struct { 3 usages  new *
    ID          int64      `gorm:"primary_key;column:id;autoIncrement"`
    UserId       string      `gorm:"column:user_id"`
    Address      string      `gorm:"column:address"`
    CreatedAt    time.Time  `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time  `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    User         User       `gorm:"foreignKey:user_id;references:id"`
}
```



Kode : Preload atau Join

```
func TestBelongsTo(t *testing.T) { new *
    fmt.Println("Preload")
    var addresses []Address
    err := db.Preload("User").Find(&addresses).Error
    assert.Nil(t, err)

    fmt.Println("Joins")
    addresses = []Address{}
    err = db.Joins("User").Find(&addresses).Error
    assert.Nil(t, err)
}
```



Belongs To di One to One

- Selain di One to Many, sebenarnya Belongs To bisa diimplementasikan di relasi One to One
- Sebelumnya kita tahu bahwa User punya satu (Has One) Wallet, artinya Wallet itu milik (Belongs To) User
- Kita bisa tambahkan field User di Wallet sebagai relasi Belongs To
- Namun, karena di Golang, cyclic itu tidak boleh, maka untuk menambahkan relasi Belongs To di One to One, kita perlu menggunakan Pointer



Kode : Wallet Model

```
▼ type Wallet struct { 8 usages  new *
    ID      string    `gorm:"primary_key;column:id"`
    UserId   string    `gorm:"column:user_id"`
    Balance  int64     `gorm:"column:balance"`
    CreatedAt time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    User     *User     `gorm:"foreignKey:user_id;references:id"`
    // gunakan pointer untuk menghindari cyclic dependency
}
```



Kode : Belongs To di One to One

```
func TestBelongsToOneToOne(t *testing.T) { new *
    fmt.Println("Preload")
    var wallets []Wallet
    err := db.Preload("User").Find(&wallets).Error
    assert.Nil(t, err)

    fmt.Println("Joins")
    wallets = []Wallet{}
    err = db.Joins("User").Find(&wallets).Error
    assert.Nil(t, err)
}
```

Many to Many



Many to Many

- Relasi yang paling kompleks ada relasi Many to Many
- Seperti yang kita tahu, bahwa untuk relasi Many to Many, kita harus membuat tabel jembatan penghubung antara dua tabel
- GORM juga mendukung relasi Many to Many, caranya mudah kita hanya perlu membuat field berupa Slice di kedua Model yang berelasi
- Untuk memberitahu tabel penghubung dan juga kolom untuk join nya, kita bisa menggunakan tag



Many to Many Tag

- Untuk memberitahu nama tabel penghubung, kita bisa menggunakan tag `gorm:"many2many:nama_tabel"`
- Saat melakukan query Many to Many, terdapat banyak sekali kolom yang perlu diketahui, seperti kolom id di Model pertama, kolom foreign key Model pertama di tabel penghubung, kolom id di Model kedua, kolom foreign key Model kedua di tabel penghubung, semua bisa kita gunakan tag
- `gorm:"foreignKey:kolom_id"` untuk id di Model pertama
- `gorm:"joinForeignKey:kolom_id"` untuk foreign key Model pertama di tabel penghubung
- `gorm:"references:kolom_id"` untuk id di Model kedua
- `gorm:"referencesForeignKey:kolom_id"` untuk foreign key Model kedua di tabel penghubung



Contoh Kasus

- Misal kita akan membuat fitur dimana User bisa like banyak Product, dan satu Product bisa di like banyak User
- Artinya kita akan membuat model Product, lalu akan membuat Relasi Many to Many antara User dan Product



Kode : Tabel Products

```
create table products
(
    id            varchar(100) not null,
    name          varchar(100) not null,
    price         bigint       not null,
    created_at    timestamp    not null default current_timestamp,
    updated_at    timestamp    not null default current_timestamp on update current_timestamp,
    primary key (id)
) engine = InnoDB;
```

Kode : Product Model

```
type Product struct { 1 usage new *
    ID      string    `gorm:"primary_key;column:id"`
    Name     string    `gorm:"column:name"`
    Price    int64     `gorm:"column:price"`
    CreatedAt time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (p *Product) TableName() string { new *
    return "products"
}
```



Kode : Tabel Penghubung user_like_product

```
✓ < create table user_like_product
  < (
    user_id    varchar(100) not null,
    product_id varchar(100) not null,
    primary key (user_id, product_id),
    foreign key (user_id) references users (id),
    foreign key (product_id) references products (id)
  ) engine = InnoDB;
```

Kode : User Model

```
type User struct { 47 usages new *
    ID          string    `gorm:"primary_key;column:id;< -:create"`
    Password    string    `gorm:"column:password"`
    Name        Name      `gorm:"embedded"`
    CreatedAt   time.Time `gorm:"column:created_at;autoCreateTime;< -:create"`
    UpdatedAt   time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    Information string    `gorm:"- "`
    Wallet      Wallet    `gorm:"foreignKey:user_id;references:id"`
    Addresses   []Address `gorm:"foreignKey:user_id;references:id"`
    LikeProducts []Product `gorm:"many2many:user_like_product;foreignKey:id;joinForeignKey:user_id;references:id;joinReferences:product_id"`
}
```



Kode : Product Model

```
type Product struct { 2 usages new *
    ID          string    `gorm:"primary_key;column:id"`
    Name        string    `gorm:"column:name"`
    Price        int64     `gorm:"column:price"`
    CreatedAt    time.Time `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt    time.Time `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
    LikedByUsers []User    `gorm:"many2many:user_like_product;foreignKey:id;joinForeignKey:product_id;references:id;joinReferences:user_id"`
}

func (p *Product) TableName() string { new *
    return "products"
}
```




Create/Update/Delete Relasi Many to Many

- Salah satu tantangan relasi Many to Many adalah, bagaimana cara melakukan Create/Update/Delete relasi Many to Many?
- Hal ini karena tabel penghubung tidak memiliki representasi Model di GORM
- Untungnya, GORM bisa digunakan untuk memanipulasi data, tanpa harus menggunakan Model
- Kita bisa gunakan method Table() pada form.DB untuk menyebut tabel mana yang mau kita pilih
- Walaupun cara ini bisa dilakukan, tapi sebenarnya cara yang lebih baik itu menggunakan fitur bernama Association Mode, yang akan kita bahas di materi selanjutnya



Kode : Create Many to Many

```
func TestCreateManyToMany(t *testing.T) { new *
    product := Product{
        ID:      "P001",
        Name:     "Contoh Product",
        Price:    10000000,
    }
    err := db.Create(&product).Error
    assert.Nil(t, err)

    err = db.Table("user_like_product").Create(map[string]interface{}{
        "user_id":    "1",
        "product_id": "P001",
    }).Error
    assert.Nil(t, err)

    err = db.Table("user_like_product").Create(map[string]interface{}{
        "user_id":    "2",
        "product_id": "P001",
    }).Error
    assert.Nil(t, err)
}
```



Kode : Preload

```
▼ func TestPreloadManyToMany(t *testing.T) { new *
    var product Product
    err := db.Preload("LikedByUsers").First(&product, "id = ?", "P001").Error
    assert.Nil(t, err)
    assert.Equal(t, 2, len(product.LikedByUsers))
}💡
```

Association Mode



Association Mode

- GORM memiliki fitur bernama Association Mode, yang digunakan untuk memanipulasi data relasi menggunakan object `gorm.Association`
- Fitur ini sangat berguna, misal ketika kita ingin menambah relasi ke Model, terutama untuk relasi Many To Many yang tidak memiliki Model untuk tabel penghubungnya
- Untuk membuat object Association, kita cukup menggunakan method `Association(relasi)`



Kode : Mencari Relasi

```
func TestAssociationFind(t *testing.T) { new *
    var product Product
    err := db.First(&product, "id = ?", "P001").Error
    assert.Nil(t, err)

    var users []User
    err = db.Model(&product).Where("first_name LIKE ?", "User%").Association("LikedByUsers").Find(&users)
    assert.Nil(t, err)
    assert.Equal(t, 1, len(users))
}
```



Kode : Add Relation

```
✓ func TestAssociationAdd(t *testing.T) { new *  
    var user User  
    err := db.First(&user, "id = ?", "3").Error  
    assert.Nil(t, err)  
  
    var product Product  
    err = db.First(&product, "id = ?", "P001").Error  
    assert.Nil(t, err)  
  
    err = db.Model(&product).Association("LikedByUsers").Append(&user)  
    assert.Nil(t, err)  
}
```



Kode : Replace Relation

```
func TestAssociationReplace(t *testing.T) { new *
    err := db.Transaction(func(tx *gorm.DB) error {
        var user User
        err := tx.First(&user, "id = ?", "1").Error
        assert.Nil(t, err)

        wallet := Wallet{
            ID:      "01",
            UserId:  "1",
            Balance: 1000000,
        }
        err = tx.Model(&user).Association("Wallet").Replace(&wallet)
        return err
    })
    assert.Nil(t, err)
}
```




Kenapa Error?

- Hal ini karena di Tabel wallets, kita menambah aturan constraint NOT NULL
- Jadi ketika GORM membuat wallet dengan id 01, dan dia coba menghapus user_id di wallet sebelumnya yaitu wallet 1, maka terjadi error
- Namun dari sini kita bisa tahu bahwa GORM akan menghapus relasi ke data sebelumnya ketika kita melakukan replace
- Replace() ini hanya cocok untuk relasi One to One atau Belongs To, dan ketika kita coba menggunakan Append() di relasi tersebut, secara otomatis GORM akan mengubah menjadi operasi Replace(), bukan Append() lagi



Kode : Delete Relation

```
func TestAssociationDelete(t *testing.T) { new *
    var user User
    err := db.First(&user, "id = ?", "3").Error
    assert.Nil(t, err)

    var product Product
    err = db.First(&product, "id = ?", "P001").Error
    assert.Nil(t, err)

    err = db.Model(&product).Association("LikedByUsers").Delete(&user)
    assert.Nil(t, err)
}
```



Kode : Clear Relation

```
✓ func TestAssociationClear(t *testing.T) { new *  
    var product Product  
    err := db.First(&product, "id = ?", "P001").Error  
    assert.Nil(t, err)  
  
    err = db.Model(&product).Association("LikedByUsers").Clear()  
    assert.Nil(t, err)  
}
```

Preloading



Preloading

- Sebelumnya kita sudah tahu bahwa untuk melakukan loading relasi, kita bisa menggunakan Preloading()
- Bagaimana jika kita ingin menambahkan kondisi ketika melakukan Preloading?
- Kita bisa tambahkan Inline Condition ketika melakukan Preloading



Kode : Preloading with Condition

```
✓ func TestPreloadingWithCondition(t *testing.T) { new *  
    var user User  
    err := db.Preload("Wallet", "balance > ?", 1000000).First(&user, "id = ?", "1").Error  
    assert.Nil(t, err)  
}
```



Nested Preloading

- Preloading juga bisa dilakukan untuk relasi yang Nested
- Misal kita akan melakukan query untuk model Wallet, kita ingin melakukan Preload ke User dan juga ke Address nya
- Kita bisa gunakan Preloading dengan menggunakan . (titik), misal “User.Addresses”



Kode : Nested Preloading

```
✓ func TestNestedPreloading(t *testing.T) { new *  
    var wallet Wallet  
    err := db.Preload("User.Addresses").Find(&wallet, "id = ?", "1").Error  
    assert.Nil(t, err)  
}
```




Preload All

- Jika kita ingin melakukan Preload semua relasi di Model, kita bisa menggunakan `clauses.Associations` ketika melakukan `Preloading()`
- Namun perlu diingat, bahwa Preload All tidak akan melakukan load Nested Relation



Kode : Preload All

```
▼ func TestPreloadAll(t *testing.T) { new *  
    var user User  
    err := db.Preload(clause.Associations).First(&user, "id = ?", "1").Error  
    assert.Nil(t, err)  
    }💡
```

Joins



Join Query

- Sebelumnya kita sudah melakukan Joins() dengan menyebutkan nama field Relation nya
- Selain menggunakan nama Relation, kita juga menggunakan Query manual ketika melakukan Joins
- Bedanya adalah, kolom Relation nya tidak akan di Query



Kode : Join Query

```
func TestJoinQuery(t *testing.T) { new *
    var users []User
    err := db.Joins("join wallets on wallets.user_id = users.id").Find(&users).Error
    assert.Nil(t, err)
    assert.Equal(t, 3, len(users))

    users = []User{}
    err = db.Joins("Wallet").Find(&users).Error // using left join
    assert.Nil(t, err)
    assert.Equal(t, 17, len(users))
}
```



Join Condition

- Saat menggunakan Joins, ketika kita ingin menambahkan kondisi di Join Table nya, jangan lupa untuk menyebutkan nama tabel nya
- Namun saat menggunakan Joins() menggunakan nama relasi, secara otomatis Gorm akan membuat alias untuk nama relasi, jadi kita harus menggunakan nama relasi ketika menambahkan kondisi



Kode : Join Condition

```
func TestJoinQueryCondition(t *testing.T) { new *
    var users []User
    err := db.Joins("join wallets on wallets.user_id = users.id AND wallets.balance > ?", 500000).Find(&users).Error
    assert.Nil(t, err)
    assert.Equal(t, 3, len(users))

    users = []User{}
    err = db.Joins("Wallet").Where("Wallet.balance > ?", 500000).Find(&users).Error // alias menggunakan nama field
    assert.Nil(t, err)
    assert.Equal(t, 3, len(users))
}
```

Query Aggregation



Query Aggregation

- Ada banyak Query Aggregation yang biasanya tersedia di Database
- Di GORM, hanya menyediakan method Aggregation untuk Count() saja
- Jika kita ingin melakukan Aggregation misal menggunakan Avg, Sum, Max, dan lain-lain
- Maka kita harus melakukan secara manual menggunakan Select()



Kode : Count Aggregation

```
func TestCount(t *testing.T) { new *
    var count int64
    err := db.Model(&User{}).Joins("Wallet").Where("Wallet.balance > ?", 500000).Count(&count).Error
    assert.Nil(t, err)
    assert.Equal(t, int64(3), count)
}
```



Kode : Other Aggregation

```
type AggregationResult struct { 1 usage  new *
    TotalBalance int64
    MinBalance    int64
    MaxBalance    int64
    AvgBalance    float64
}

func TestAggregation(t *testing.T) { new *
    var result AggregationResult
    err := db.Model(&Wallet{}).Select("sum(balance) as total_balance", "min(balance) as min_balance",
        "max(balance) as max_balance", "avg(balance) as avg_balance").Take(&result).Error
    assert.Nil(t, err)
    assert.Equal(t, int64(3000000), result.TotalBalance)
    assert.Equal(t, int64(1000000), result.MinBalance)
    assert.Equal(t, int64(1000000), result.MaxBalance)
    assert.Equal(t, float64(1000000), result.AvgBalance)
```



Group By & Having

- GORM juga menyediakan method untuk melakukan Group By menggunakan method `Group()`
- Dan untuk melakukan Having menggunakan method `Having()`

Kode : Group By & Having

```
func TestGroupByHaving(t *testing.T) {  
    new *  
    var result []AggregationResult  
    err := db.Model(&Wallet{}).Select("sum(balance) as total_balance", "min(balance) as min_balance",  
        "max(balance) as max_balance", "avg(balance) as avg_balance").  
        Joins("User").Group("User.id").Having("sum(balance) > ?", 1000000).  
        Find(&result).Error  
    assert.Nil(t, err)  
    assert.Equal(t, 0, len(result))  
}
```

Context



Context

- Saat kita menggunakan Golang, biasanya kita akan menggunakan `context.Context`
- Bagaimana dengan GORM? Dari awal kita belum pernah menggunakan Context
- GORM juga mendukung penggunaan Context, kita bisa menggunakan method `WithContext()` ketika mau melakukan operasi menggunakan GORM



Kode : Context

```
▼ func TestContext(t *testing.T) { new *  
    ctx := context.Background()  
  
    var users []User  
    err := db.WithContext(ctx).Find(&users).Error  
    assert.Nil(t, err)  
    assert.Equal(t, 17, len(users))  
}
```

Scopes



Scopes

- Sebelumnya kita sudah pernah menggunakan method `Scopes()`
- Method `Scopes()` juga bisa digunakan untuk menambahkan custom logic yang mungkin sering kita gunakan
- Kita cukup menambahkan function `(*gorm.DB) *gorm.DB`



Kode : Sample Function

```
▼ func BrokeWalletBalance(db *gorm.DB) *gorm.DB { no usages  new *  
    return db.Where("balance = ?", 0)  
}  
  
▼ func SultanWalletBalance(db *gorm.DB) *gorm.DB { no usages  new *  
    return db.Where("balance > ?", 1000000)  
}
```



Kode : Scopes

```
func TestScopes(t *testing.T) { new *
    var wallets []Wallet
    err := db.Scopes(BrokeWalletBalance).Find(&wallets).Error
    assert.Nil(t, err)

    wallets = []Wallet{}
    err = db.Scopes(SultanWalletBalance).Find(&wallets).Error
    assert.Nil(t, err)
}
```

Connection Pool



Connection Pool

- Saat kita belajar di kelas Golang Database, kita belajar tentang Pool, dimana Golang mengatur maintain koneksi yang terbuka dan tertutup secara otomatis
- Kita hanya cukup menggunakan saja, tanpa harus pusing mengaturnya
- Bagaimana dengan GORM?
- GORM sendiri sebenarnya didalamnya tetap menggunakan sql.DB
- Jadi jika kita ingin mengubah pengaturan Pool nya, kita bisa menggunakan sql.DB



Kode : Connection Pool

```
sqlDB, err := db.DB()
if err != nil {
    panic(err)
}
sqlDB.SetMaxOpenConns(100)
sqlDB.SetMaxIdleConns(10)
sqlDB.SetConnMaxLifetime(30 * time.Minute)
sqlDB.SetConnMaxIdleTime(5 * time.Minute)
```



Migrator



Migrator

- Sebelumnya kita pernah belajar tentang database migration di kelas Golang Database Migration
- Sebenarnya GORM sendiri memiliki fitur untuk melakukan Migration secara otomatis.
- Namun lebih direkomendasikan menggunakan database migration yang mendukung version agar tidak terjadi kesalahan ketika melakukan perubahan schema database
- Untuk mendapatkan object Migrator, kita bisa menggunakan method Migrator()
- <https://pkg.go.dev/gorm.io/gorm#Migrator>
- Kita tidak akan membahas terlalu panjang fitur Migrator ini, karena jarang sekali digunakan di aplikasi nyata, biasanya hanya digunakan untuk melakukan pengetesan di komputer programmer saja

Kode : GuestBook Model

```
type GuestBook struct {
    ID          int64    `gorm:"primary_key;column:id;autoIncrement"`
    Name        string   `gorm:"column:name"`
    Email       string   `gorm:"column:email"`
    Message     string   `gorm:"column:message"`
    CreatedAt   string   `gorm:"column:created_at;autoCreateTime"`
    UpdatedAt   string   `gorm:"column:updated_at;autoCreateTime;autoUpdateTime"`
}

func (g *GuestBook) TableName() string {
    return "guest_books"
}
```



Kode : Migrator

```
✓ func TestMigrator(t *testing.T) { new *  
    err := db.Migrator().AutoMigrate(&GuestBook{})  
    assert.Nil(t, err)  
}
```

Hook



Hook

- GORM memiliki fitur bernama Hook
- Hook adalah function di dalam Model yang akan dipanggil sebelum melakukan operasi create/query/update/delete
- Jika function Hook tersebut mengembalikan error, secara otomatis GORM akan menghentikan operasi nya
- Function Hook menggunakan struktur :
`func(*gorm.DB) error`



Hook untuk Create

```
// begin transaction  
BeforeSave()  
BeforeCreate()  
// save before associations  
// insert into database  
// save after associations  
AfterCreate()  
AfterSave()  
// commit or rollback transaction
```



Hook untuk Update

```
// begin transaction  
BeforeSave()  
BeforeUpdate()  
// save before associations  
// update database  
// save after associations  
AfterUpdate()  
AfterSave()  
// commit or rollback transaction
```



Hook untuk Delete

```
// begin transaction  
BeforeDelete()  
// delete from database  
AfterDelete()  
// commit or rollback transaction
```




Hook untuk Find

```
// load data from database  
// Preloading (eager loading)  
AfterFind()
```

Kode : User.BeforeCreate()

```
func (u *User) BeforeCreate(db *gorm.DB) error {
    if u.ID == "" {
        u.ID = "user-" + time.Now().Format("20060102150405")
    }
    return nil
}
```

```
func TestUserHook(t *testing.T) {
    user := User{
        Password: "rahasia",
        Name: Name{
            FirstName: "User 100",
        },
    }
    err := db.Create(&user).Error
    assert.Nil(t, err)
    assert.NotNil(t, user.ID)
    assert.NotEqual(t, "", user.ID)
}
```

Performance



Performance

- GORM sendiri sebenarnya sudah di optimize agar performance nya baik
- Jadi secara default GORM sudah baik jika kita gunakan untuk membuat aplikasi
- Tapi berikut kita bahas beberapa tips yang bisa kita lakukan lagi ketika ingin meningkatkan performa aplikasi kita ketika menggunakan GORM



Matikan Auto Transaction

- Secara default, ketika kita melakukan Create, Update, Delete, semua akan dijalankan dalam Transaction, walaupun kita tidak melakukannya
- Kita bisa mematikan fitur Auto Transaction ini jika kita mau
- Kita bisa ubah config ketika membuat gorm.DB dengan menambahkan SkipDefaultTransaction menjadi true
- Dengan mematikan fitur ini, kita harus melakukan transaction secara manual



Kode : Mematikan Auto Transaction

```
✓ func OpenConnection() *gorm.DB { 1 usage  new *  
    dialect := mysql.Open("root:@tcp(127.0.0.1:3306)/belajar_golang_g  
✓    db, err := gorm.Open(dialect, &gorm.Config{  
        Logger: logger.Default.LogMode(logger.Info),  
        SkipDefaultTransaction: true,  
    })  
✓    if err != nil {  
        panic(err)  
    }  
}
```



Cache Prepared Statement

- Secara default saat GORM membuat Query menggunakan Prepare Statement, dia tidak akan menyimpan SQL Query nya di memory
- Kita bisa mengaktifkan fitur ini, agar Prepare Statement disimpan di memory, sehingga ketika kita sering menggunakan Query yang sama, GORM tidak perlu membuat Prepare Statement nya lagi, cukup menggunakan yang sudah dibuat di memory
- Kita bisa aktifkan fitur ini di config dengan nama PrepareStmt



Kode : Cache Prepared Statement

```
func OpenConnection() *gorm.DB { 1 usage new *
    dialect := mysql.Open("root:@tcp(127.0.0.1:3306)/belajar_golang_gorm?ch
    db, err := gorm.Open(dialect, &gorm.Config{
        Logger:                logger.Default.LogMode(logger.Info),
        SkipDefaultTransaction: true,
        PrepareStmt: true,
    })
    if err != nil {
```




Select Fields

- Secara default, saat kita menggunakan GORM, semua field di Model akan di select
- Ketika kita hanya butuh beberapa saja, tidak disarankan untuk melakukan select semua field, apalagi memang field nya tidak kita butuhkan
- Sangat disarankan untuk memilih field mana yang mau kita select menggunakan method `Select()`
- Seperti yang pernah kita bahas di materi-materi sebelumnya



Large Result

- Saat kita melakukan query Find() ke Slice, secara default GORM akan mengambil seluruh data dan menyimpannya di dalam Slice
- Baik semua isi Slice mau kita baca atau tidak, hal ini kadang tidak optimal jika ukuran hasil query nya besar
- Disarankan untuk menggunakan Lazy Result menggunakan Rows() jika memang kita ingin melakukan Query dengan jumlah hasil yang besar, sehingga kita bisa ambil data yang dibutuhkan saja, tanpa harus menyimpan semuanya ke memory



Table Split

- Jika Model yang kita buat terlalu banyak field nya, maka secara default semua field akan di Query oleh GORM
- Pada kasus seperti ini, selain kita bisa Select field nya satu per satu, kita juga bisa coba Split Model nya menjadi relasi One to One
- Sehingga kita cukup Query pada data yang kita butuhkan, contoh sebelumnya kita melakukan Split Tabel antara Model User dan Model Wallet

Penutup