

Framework yaitu kerangka kerja yang sudah disiapkan oleh ahli nya, framework kita gunakan untuk membangun isinya, biar kerjaan code kita terstruktur

BELAJAR BASIC PROGRAMMING

Jika menggunakan

1. NODE JS sering disebut mern stack singkatan dari mongoDB Express React dan node js, framework yang digunakan untuk backend nya adalah express Js
2. Php memakai framework Laravel
3. JAVA Framework Springboot
4. Python Framework Django dan flask
5. Golang Framework Gin, echo

VERSION CONTROL

-git clone: clone repository

- git status: status perubahan
- git add nama file dan git add . untuk semua file : menambahkan file ke staggig area
- git commit -m "pesan perubahan" : commit perubahan
- git push origin nama branch : push ke remote

Buka pull request do github untuk review

- git init: membuat repo baru di folder local
- git log: melihat Riwayat commit
- git diff: melihat perubahan detail
- git checkout "branch" : pindah ke branch lain
- git merge "branch" : gabungkan branch
- git pull: sinkronisasi dari github ke local
- git reset -hard : kembalikan repo ke commit tertentu
- git stash : simpan sementara perubahn yang belum di commit

* Contoh Skenario Lengkap

```
bash

# kamu sedang di branch main
git checkout main

# buat branch baru untuk fitur login
git checkout -b feature/login

# ubah kode dan commit
git add .
git commit -m "Add login API endpoint"

# kirim ke GitHub
git push origin feature/login
```

Lalu di GitHub:

- Buat **Pull Request** dari `feature/login` → ke `develop`
- Setelah disetujui → merge
- Setelah semua fitur stabil di `develop` → merge ke `main` untuk rilis

FUNDAMENTAL DATABASE

Secara umum 2 jenis database yaitu

SQL dan noSQL

Sql = mysql, postgresql

NoSql = Mongodb

Cari tau kapan menggunakan sql dan nosql

Alur koneksi database

1. Koneksi sql server
2. Membuat file.env yang berisi db name db host db password db user , isi semua dapat dilihat dari pertama kita membuat konesi database
3. Setelah itu membuat konfigurasi agar si .env dapat berjalan yaitu file pkg berisi config didalamnya config dan database, di config saya menggunakan libarry viper untuk membaca .env , Hanya berupa **struct data** tanpa logic koneksi mulai dari membuat stuct type config kemudian function config disinila viper digunakan bukan hanya itu untuk konfigurasi database viper juga digunakan untuk konfigurasi

- ServiceHost → alamat server backend (misal: localhost)
- ServicePort → port service kamu (misal: 8080)
- ServiceEndpointV → versi API (misal: /api/v1)
- ServiceEnvironment → environment (misal: development atau production)

Setelah mengenai config maka ke **database config**, di config kita sudah buat viper konfigurasi untuk membaca .env tadi dan di database config ini di dalamnya menampung type database seperti mysql postgresql dan juga redis lalu membuat struct membedakan koneksi read dan write database database config ini bertanggung jawab membuka koneksi nyata ke database Menggunakan GORM

4. Di package pkg/database itu saya bagi menjadi tiga file agar koneksi database lebih terstruktur
- **db.go** berisi definisi tipe dan konfigurasi dasar, seperti tipe database (Postgres, MySQL, dll), struktur Args untuk menampung konfigurasi koneksi (username, password, host, port, schema, dll), dan fungsi IsValid() untuk memastikan parameter koneksi benar
 - **url.go** bertugas membentuk *connection string* atau URL database berdasarkan konfigurasi di Args. Jadi file ini yang menyusun format seperti postgres://user:password@host:port/dbname?sslmode=disable. Fungsi utamanya GetURL() dan GetURLString() — supaya conn.go bisa langsung pakai tanpa menulis ulang logika pembuatan URL.
 - **conn.go** adalah inti koneksi — file ini membuka koneksi ke database menggunakan GORM. Fungsi GetConnection() akan memeriksa parameter, menentukan driver (postgres.Open() atau mysql.Open()), lalu mengembalikan *gorm.DB. Selain itu ada GetReadWriteConnection() kalau ingin pisahkan koneksi baca dan tulis (read/write separation).

Di pkg database ini akan ada GetConnection()

Kemudian di app go menjalankan GetConnection() GORM terkoneksi ke SQL Server/Postgres/MySQL

PKG CUSTOMERRORS

saya membuat customerrors pada pkg yaitu untuk membuat error handling yang dimana yang saya buat itu yaitu

1. bad request yaitu Error 400 terjadi ketika server tidak bisa memproses permintaan dari client karena ada kesalahan pada data atau format request.

Contoh: Mengirim data JSON yang salah format ke API.

dicode saya membuat singkatnya seperti

- var ErrBadRequest = errors.New("bad request") Golang
- throw new RuntimeException("Bad request"); - Java
- raise Exception("Bad request") - python

2. Error 404 (Not Found)

Terjadi ketika client meminta resource yang tidak ada di server.

Contoh: Mengakses URL /users/123 padahal user dengan ID 123 tidak ada. saya membuat singkatnya seperti TrackableError untuk error tersebut dapat dilacak

- var ErrNotFound = &TrackableError{Code:404, Message:"Not Found"} Go
- throw new NotFoundError("User ID 123 tidak ditemukan"); Java
- raise NotFoundError("User ID 123 tidak ditemukan") python

3. Error 500 (Internal Server Error)

Terjadi ketika server mengalami masalah di sisi backend, biasanya karena bug atau kesalahan konfigurasi server.

Contoh: Terjadi crash pada aplikasi server atau database gagal diakses.

- var ErrInternalServerError = errors.New("internal server error") GO
- throw new RuntimeException("Internal server error"); JAVA
- raise Exception("Internal server error") PYTHON

Implementasi error handling tersebut yaitu di repository untuk 400 dan 500 karena 400 data tidak ada dan 500 ketika database error

```
user, err := userRepo.FindByID("123")
```

dan Handle business logic errors (400) di Usercase karena invalide data dan duplicate untuk 409

```
err := userUsecase.ValidateUser(data)
```

dan di controller di `statusCode := customerror.GetStatusCode(err)` untuk auto mapping error → HTTP status

APP

INIT

init.go

- Mengumpulkan semua dependencies (database, config, redis)
- Membuat semua instances (repositories, usecases, controllers)

gunanya untuk lebih simple instances dan ada unit test go UALITY TEST yang memastikan factory bekerja benar.

jika tidak menggunakan init ribet contoh `clientRepo := &clientRepository{db: database, config: config, redis: redis}`

Constants

jadi konstants ini ibarat nya hanya pesan ketika uuid atau id tidak ditemuka dan tidak valid kata lain Hanya berisi template pesan contoh, client yang dibuat double maka pesannya `ErrDuplicateClient = `Duplicate client! This client is already registered.``` ini diimplementasikan di usecase memanggil repository untuk cek data return error dengan pesan dari constants tadi

memastikan pesan ini benar benar valid dilakukan validasi di usecase kemudian di implementasikan di controller

contoh kita buat pada usecase validasi `IsValidClientUuid` yang Dimana MEMASTIKAN CLIENT DENGAN UUID TERSEBUT EXISTS DI DATABASE kemudian digunakan di controller panggil usecase untuk validasi terima error dari usecase dan kirim response ke client

Helper

Mapping yaitu konversi data map string interface
contoh:

- string To Int mengubah string ke integer
- ToSliceString(key string, source map[string]interface{}) Mengubah value di map menjadi **slice of interface**, khususnya untuk string atau int. Kenapa perlu: Kadang API atau DB butuh data berbentuk slice.

Response helper

response API, supaya semua response dari backend konsisten formatnya. Membantu debugging karena response nya konsisten

helpers = BUNGKUS (format response JSON)

constants = ISI (pesan success/error) 

REPOSITORY Interaksi dengan database menggunakan model

Contoh pertama buat type client interface type

```
ClientInterface interface {
```

```
    Create(client MasterClient) error      // ← SIMPAN data
    GetByUUID(uuid string) MasterClient  // ← CARI data
    Update(client MasterClient) error     // ← UPDATE data
    Delete(uuid string) error           // ← HAPUS data
    Search(filter) []MasterClient       // ← CARI dengan filter
```

```
}
```

implementasi

```
func (r *repo) Create(client MasterClient) error {
```

```
    // ↓ TUGAS REPOSITORY: SIMPAN KE DATABASE
```

```
    err := r.db.Create(&client).Error
```

```
    return err
```

```
}
```

```
func (r *repo) GetByUUID(uuid string) MasterClient {  
    // ↓ TUGAS REPOSITORY: CARI DI DATABASE  
    var client MasterClient  
    r.db.Where("uuid = ?", uuid).First(&client)  
    return client  
}
```

USECASE

menjalankan logika bisnis, mengambil/menyimpan data lewat **repository**.

```
// repository/client_repository.go  
  
func (r *repo) CreateMasterClient(client models.MasterClient) (models.MasterClient, error) {  
  
    err := r.Options.Postgres.Create(&client).Error // ← URUS DATABASE  
  
    return client, err  
}
```

Controller

```
memproses request, memanggil usecase  
  
// controller/client_controller.go  
  
func (c *controller) Create(ctx echo.Context) error {  
  
    // BIND REQUEST  
  
    var req models.CreateClientRequest  
  
    ctx.Bind(&req)
```

```

// PANGGIL USECASE
client, err := c.Options.UseCases.Client.CreateMasterClient(req)

// FORMAT RESPONSE
return helpers.StandardResponse(ctx, 201, "Success", client, nil)
}

```

Routes

menerima request, mengarahkannya ke **controller**.

STRUKTUR URL:			
HTTP Method	URL Pattern	Controller Method	Kegunaan
POST	/v1/master-client	Client.Create	Buat client baru
GET	/v1/master-client	Client.Get	Lihat semua client
GET	/v1/master-client/:uuid	Client.GetDetail	Lihat detail client
PATCH	/v1/master-client/:uuid	Client.Update	Update client
DELETE	/v1/master-client/:uuid	Client.Delete	Hapus client

- Saat Anda menulis handler/controller dengan anotasi tertentu, tools seperti swaggo akan menghasilkan file di folder docs.
- **Integrasi dengan server:** Biasanya, aplikasi backend akan meng-serve dokumentasi Swagger di endpoint khusus (misal: /swagger/index.html).
- **Kebutuhan utama:**
 - Transparansi API
 - Testing API
 - Onboarding developer baru
 - Memastikan API sesuai spesifikasi

MENDESAIN API

1. Paham prinsip RestfulAPI

Jawab

- Penamaan pada setiap sintaks yang baik contoh
/master-client/{uuid}/workhour bukan seperti /getClientById
- Proper HTTP Methods:

```
POST  /master-client          // ↴ Create client
GET   /master-client          // ↴ Get all clients
GET   /master-client/{uuid}    // ↴ Get specific client
PATCH /master-client/{uuid}    // ↴ Update client
DELETE /master-client/{uuid}   // ↴ Delete client
```

- Response structure yang bersih

```
// Success response
{
  "status_code": 200,
  "message": "Success",
  "data": { ... }
}

// Error response
{
  "status_code": 404,
  "message": "Client with id '123' not found"
}
```

2. keamanan API, authentikasi, otorisasi menggunakan JWT

fungsi: Menandai user sudah login dan valid. Token dikirim di header Authorization: Bearer <token> tiap request ke API.

Saya buat JWT dengan secret key dari .env, generate token saat login berisi user info & roles, pakai middleware untuk validate token setiap request, dan cek roles untuk authorization. Semua route protected pakai middleware agar akses aman dan sesuai hak user."

Token dikirim di header Authorization: Bearer <token> setiap request, baik di Postman saat testing maupun di frontend/mobile app saat memanggil API.

3. Seceruty dan testing

Jawab:

untuk security saya menggunakan layer yaitu

- Layer 1: "Semua input dari client divalidasi di 3 level:

CONTROLLER: Validasi format (required fields, email format, dll)

USECASE: Validasi business rules (unique data, constraints)

REPOSITORY: Database constraints (unique keys, foreign keys)

- layer 2: uthentication pakai JWT tokens

Authorization pakai Role-Based Access Control (RBAC):

- Setiap user punya roles (admin, manager, user)

- Middleware cek roles sebelum akses endpoint

- Context propagation: user info available di seluruh request chain

Testing

Saya fokus test USECASE layer karena ini inti business logic.

Mock dependencies (repository) untuk test isolasi.

Cover semua business rules dan error scenarios.

Contoh: "Test bahwa create client harus reject duplicate email"

4. Deployment CI/CD

Contoh,

- aplikasi relatif sederhana seperti website CV maka bisa di shared hosting atau vps, biaya terjangkau,
- aplikasi dengan framework modern seperti frontend react atau next js maka deploy di vercel dan jika laravel maka di laravel cloud
- deploy aplikasi besar memakai aws, google cloud platform, azure

CICD otomasi perubahan code bisa menggunakan github actions

"File deploy.yml adalah pipeline CI/CD otomatis di GitHub Actions. Caranya adalah: developer push code ke branch development, pipeline akan jalan otomatis; pertama menjalankan unit dan API tests (CI), lalu build Docker image dari code + .env dari GitHub Secrets, push ke registry (Build), dan terakhir update deployment di AWS EKS serta rollout aplikasi (CD). Semua variabel sensitif diambil dari GitHub Secrets, sementara code dan konfigurasi deployment berasal dari repo dan env vars."

membuat docker file

"Dockerfile saya buat sendiri pakai multi-stage build: stage pertama compile Go binary, stage kedua pakai Alpine untuk runtime agar image ringan, copy binary + env, set port dan entrypoint. Ini memastikan aplikasi siap dijalankan di container dengan ukuran minimal."

"Dockerfile saya bikin dua stage: pertama compile Go jadi binary, kedua jalankan binary di container ringan dengan Alpine, lengkap dengan file .env dan port yang dibuka."

"Pipeline CI/CD saya berjalan: pertama CI test code supaya aman, kemudian build Docker image di Ubuntu runner, push image ke registry, dan terakhir CD update deployment di cluster seperti AWS EKS supaya aplikasi langsung live."

"Saya deploy ke Kubernetes dengan cara: build Docker image, push ke registry, buat manifest YAML untuk Deployment dan Service, lalu apply ke cluster menggunakan kubectl. Di pipeline CI/CD, langkah ini otomatis sehingga setiap update code langsung rollout ke cluster."

"Saya implementasi monitoring backend dengan Prometheus dan Grafana secara end-to-end:

1. **pkg/metrics/** → buat central metrics registry (HTTP, DB, business, custom metrics)
 2. **pkg/middleware/** → middleware untuk auto-track setiap API request (status, path, response time)
 3. **app/repositories/** → instrument database operations (query performance, error rate, business ops)
 4. **routes/** → expose endpoint /metrics supaya Prometheus bisa scrape
 5. **app.go** → register middleware dan metrics collector ke Echo router
 6. **docker-compose.yml** → setup Prometheus + Grafana + aplikasi container
 7. **monitoring/** → konfigurasi Prometheus dan dashboards Grafana untuk visualisasi real-time.
- Dengan setup ini, semua metrics aplikasi bisa dipantau secara otomatis dan real-time."

REDIS

1. SETUP PERTAMA di: CONFIGURATION LAYER

"Saya setup Redis configuration terlebih dahulu:"

- **Baca config** dari environment variables (.env file)
- **Setup connection details** - host, port, password, database number
- **Siapkan Redis client** dengan proper connection pooling

2. KEMUDIAN di: INFRASTRUCTURE LAYER

"Saya buat Redis client wrapper:"

- **Handle connection management** - connect, disconnect, health check
- **Basic operations** - get, set, delete, expire

- **JSON serialization** - auto convert data ke JSON untuk storage
-

3. SELANJUTNYA di: REPOSITORY LAYER

"Saya implement caching logic di data access layer:"

Untuk Store Data:

- **Cache individual stores** - "store:123"
- **Cache store lists** - "stores:page:1:limit:10"
- **Cache store + client info** - "store_with_client:123"

Cache Strategy:

- **Check cache dulu** - jika ada, return dari cache
 - **Jika tidak ada** - ambil dari database, simpan ke cache
 - **Auto invalidate** - hapus cache ketika data di-update/dihapus
-

4. LALU di: BUSINESS LOGIC LAYER

"Saya expose cache-enabled methods:"

- **Pilih methods yang critical** - frequently accessed data
 - **Fallback mechanism** - jika cache down, tetap work dengan database
 - **Performance optimization** - tanpa ubah business logic
-

5. TERAKHIR di: API LAYER

"Saya berikan flexibility ke client:"

- **Dual endpoints** - dengan cache vs tanpa cache
- **Cache indication** - header untuk tau response dari cache atau data

"Untuk integrasi RESTful API, pertama saya pastikan endpoint API, method (GET, POST, dll), dan format request/response (JSON) sudah jelas.

Di backend saya menggunakan HTTP client (misal `net/http` di Go atau library lain) untuk melakukan request ke API eksternal.

Langkahnya:

1. "Kirim request ke endpoint API dengan header dan body sesuai dokumentasi."
2. "Terima response dan cek status code (200, 400, 500, dll)."
3. "Parse response JSON ke struct atau map agar bisa dipakai di aplikasi."
4. "Error handling: menangani timeout, response error, atau network error."
5. "Integrasi ke business logic: data yang diterima diproses, disimpan di database, atau dikembalikan ke client sesuai kebutuhan."

Saya juga biasanya menambahkan retries, logging, dan timeout supaya integrasi lebih aman dan reliable."



Jawaban interview-ready:

"Dalam mengelola dan mengoptimalkan database, pendekatan saya sistematis:

1. "Desain schema efisien → normalisasi/denormalisasi sesuai kebutuhan query."
2. "Indexing & query optimization → buat index pada kolom yang sering digunakan di WHERE, JOIN, ORDER BY; analisis query plan untuk mengurangi bottleneck."
3. "Parameterisasi & prepared statements → mencegah SQL injection dan meningkatkan performance."
4. "Connection management → gunakan connection pooling agar load database terdistribusi dengan baik."
5. "Monitoring & tuning → cek slow queries, locking, deadlocks, dan lakukan optimasi rutin."
6. "Backup & recovery → schedule backup dan test restore untuk keamanan data."

Dengan ini, database tetap cepat, aman, dan scalable."