

## BAB VII

### *SERVO CONTROLLER (BACKEND - GOLANG)*

Pada praktikum MCS bab 7, praktikan akan membangun RESTFUL API yang digunakan untuk menggerakkan servo. RESTFUL API akan melakukan pemantauan terhadap data yang tersimpan pada *database*. Jika terdapat perubahan data, maka *microcontroller* akan menghasilkan *output* untuk menggerakkan servo.

#### 7.1 Tujuan Praktikum

Tujuan	Penjelasan
Membangun <i>database</i> dengan bahasa pemrograman Golang	Dalam bab ini, praktikan akan diajarkan cara membuat sebuah <i>database</i> dengan menggunakan bahasa pemrograman Golang
Memahami cara memantau <i>database</i> dari perubahan kondisi servo	Pada bab ini, <i>database</i> digunakan sebagai media untuk memantau perubahan kondisi servo

#### 7.2 Persyaratan Praktikum

Disarankan praktikan menggunakan *hardware* dan *software* sesuai pada dokumentasi ini. Apabila terdapat versi yang lumayan lampau dari versi yang direkomendasikan atau *hardware* yang lawas maka sebaiknya bertanya kepada Asisten Mengajar Shift.

<i>HARDWARE YANG DIBUTUHKAN PRAKTIKUM</i>	JENIS
PC / Laptop CPU	≥ 4 Cores
PC / Laptop RAM	≥ 8 GB
PC / Laptop Storage	≥ 10 GB

## SOFTWARE YANG DIBUTUHKAN PRAKTIKUM

*Visual Studio Code*

**Postgre SQL**

*Postman*

### 7.3 Materi Praktikum

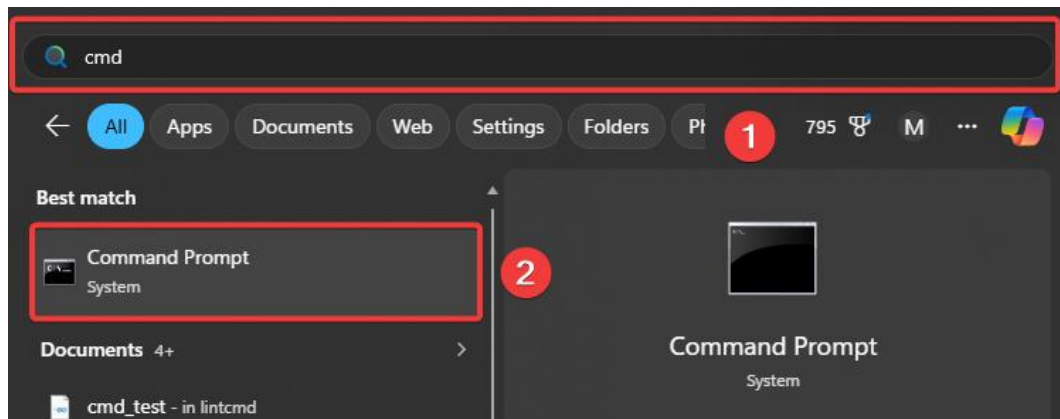
Pada bab ini aplikasi *backend* yang akan dibangun menggunakan bahasa pemrograman Go dengan *framework* yang bernama Gin. Adapun dari sisi IoT menggunakan *microcontroller* ESP32 dan *servo*. *Servo* akan bergerak sesuai dengan id yang tersimpan dalam *database*. Pada bagian *database*, *field* id akan bernilai 1 dan tidak ada data id lain yang terbentuk, Sedangkan, untuk servo status akan berisi angka 0 atau 1 dimana angka tersebut akan digunakan pada pengkondisian untuk membuat servo dapat bergerak.

### 7.4 Prosedur Praktikum

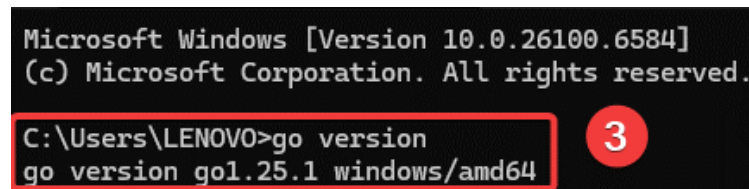
Dalam membangun REST API pada praktikum ini, terdapat beberapa langkah yang harus dilalui terlebih dahulu, sebelum nantinya melakukan pembuatan kode untuk REST API. Berikut merupakan langkah-langkah yang harus dilalui:

1. Memeriksa seluruh kebutuhan yang diperlukan
  - **Bahasa pemrograman Golang** (*Version 1.23* atau di atasnya)
  - *Visual studio code* (*Extension* Golang dan *code runner*)
  - **Postgre SQL**
  - **Postman**

Untuk memastikan apakah bahasa pemrograman golang telah terinstall pada perangkat, bukalah *command prompt* dan ketikkan perintah **go version**. Jika perangkat telah terinstall dengan bahasa golang, maka tampilan dari *command prompt* akan terlihat, seperti pada Gambar 7.1.

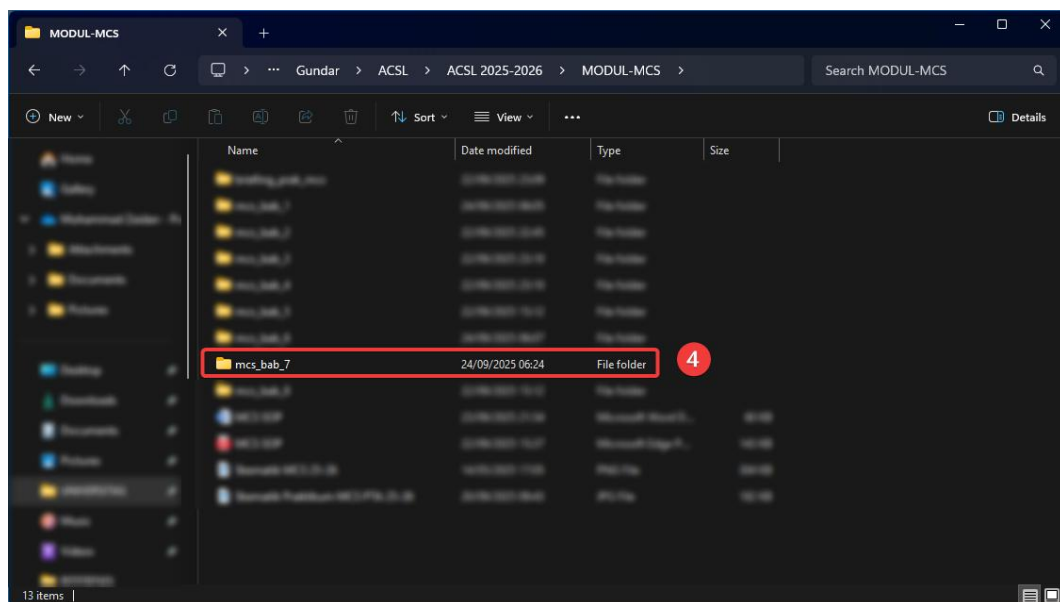


Gambar 7.1 Proses Pengecekan Golang



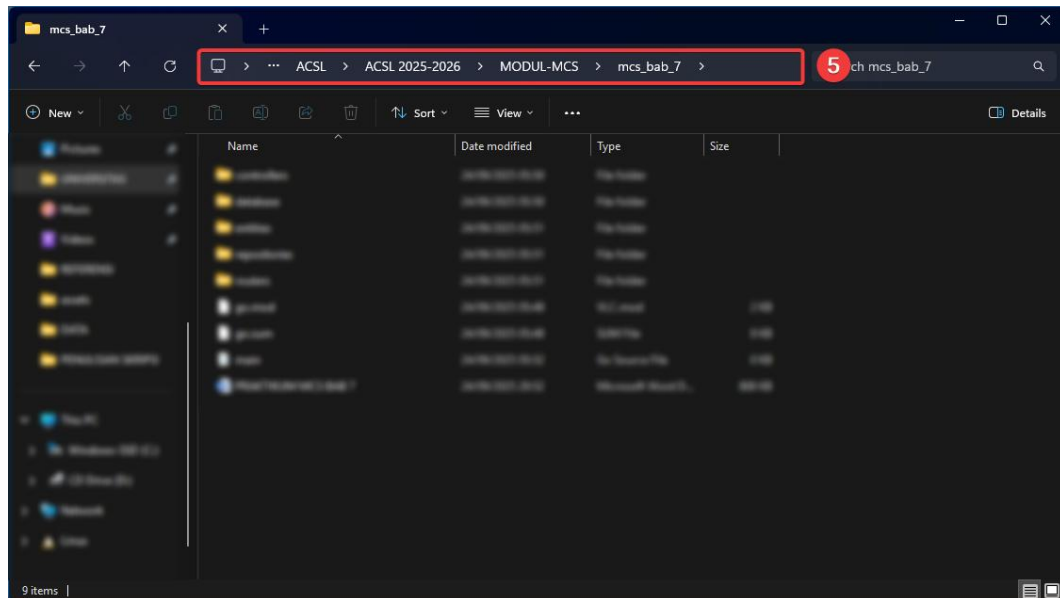
Gambar 7.2 Hasil Pengecekan Versi Golang

2. Buatlah sebuah *folder* baru dengan nama bebas. Jika nama folder lebih dari 1 suku kata, pisahkan dengan menggunakan *underscore* (\_).



Gambar 7.3 Proses Pembuatan *Folder Project*

3. Masuklah ke dalam *folder* tersebut dan ketiklah perintah **cmd** pada bagian *path folder* agar langsung masuk ke dalam *command prompt* untuk melakukan konfigurasi lebih lanjut.



Gambar 7.4 Proses Konfigurasi *Project*

4. Setelah masuk ke dalam *command prompt*, masukkan seluruh konfigurasi berikut secara satu per satu.

```
go mod init [nama_project]

go get -u "github.com/gin-gonic/gin"
go get -u "github.com/lib/pq"
go get -u "github.com/rubenv/sql-migrate"
go get -u "github.com/joho/godotenv"
```

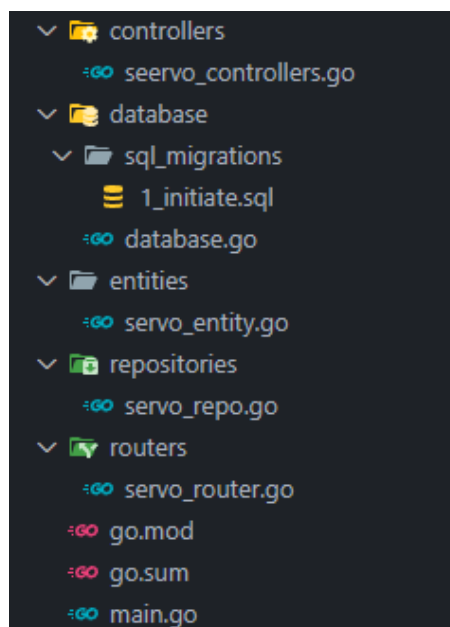
Berikut merupakan penjelasan singkat terkait kode konfigurasi yang telah dimasukkan:

1. Perintah **go mod init [nama\_project]** digunakan untuk menginisialisasi golang pada *folder project*. Hasil dari proses ini akan menghasilkan sebuah file bernama **go.mod** yang berisikan konfigurasi.
2. Perintah **go get -u "github.com/gin-gonic/gin"** digunakan untuk instalasi *package Gin framework*. *Gin framework* memudahkan pengembangan

API, karena *package* ini menyediakan berbagai fitur seperti routing, *middleware* dan *handling* JSON.

3. Perintah **go get -u "github.com/lib/pq"** digunakan untuk mengunduh / instalasi *driver* untuk PostgreSQL. *Package* tersebut digunakan agar bahasa pemrograman Go berkomunikasi dengan PostgreSQL dan mengirim *query*.
4. Perintah **go get -u "github.com/rubenv/sql-migrate"** digunakan untuk mengunduh / instalasi migrasi sql. Dengan adanya *package* ini pengembang dapat mengelola konfigurasi *database*.
5. Perintah **go get -u "github.com/joho/godotenv"** digunakan untuk mengunduh / instalasi godotenv yang digunakan untuk membaca file .env yang berisikan berbagai konfigurasi.

Setelah melakukan konfigurasi pada *project* golang, bukalah *folder* tersebut pada *software visual studio code* dan bentuklah *tree project*, seperti yang terlihat pada Gambar 7.5.



Gambar 7.5 Struktur *Tree Project*

Setelah membentuk struktur *tree project*, bukalah file **card\_bridge\_entity.go** dan masukkanlah kode program berikut:

```
package entities

type Status struct {
    Id          int `json:"id"`
    SrvStatus int `json:"srv_status"`
}
```

Kode program tersebut berperan sebagai model yang mendefinisikan variabel serta tipe data yang digunakannya. Pada praktikum kali ini, variabel yang didefinisikan ada sebanyak 2, yakni **Id** dan **SrvStatus** yang sama-sama bertipe data integer. Masing-masing variabel memiliki bentuk JSONnya sendiri, dimana data dari variabel *Id* akan dikonversi ke *key id*, sedangkan data pada *SrvStatus* akan dikonversi ke dalam *key srv\_status*.

Kemudian, bukalah file **1\_initiate.sql** yang tersimpan di dalam *folder sql\_migrations* dan masukkanlah kode program berikut:

```
-- +migrate Up
-- +migrate StatementBegin

CREATE TABLE servo (
    id INTEGER PRIMARY KEY,
    srv_status INTEGER
);

-- +migrate StatementEnd
```

Kode program di atas digunakan untuk membuat tabel *database* baru bernama **servo**. Tabel yang dibuat pada praktikum kali ini memiliki 2 *field* bernama **id** dan **srv\_status** yang bertipe data integer. *Field id* bersifat PRIMARY KEY yang artinya data pada *field* tersebut tidak dapat terduplikasi atau bersifat unik. *Field* tersebut dijadikan sebagai patokan dalam pengubahan data pada *field* *srv\_satus*. *Field* *srv\_status* nantinya akan berisikan data 0 atau 1 yang akan memiliki kondisi tertentu.

**Migrate up** merupakan instruksi yang akan menerapkan semua *query* SQL ke yang lebih baru. **Statement begin** merupakan instruksi yang menandakan

awal dari proses pembuatan *database*, sedangkan *statement end* merupakan instruksi yang menandakan akhir dari pembuatan *database*.

Berikutnya bukalah file **database.go** yang tersimpan pada folder *database* dan ketiklah kode program berikut:

```
package database

import (
    "database/sql"
    "embed"
    "fmt"

    migrate "github.com/rubenv/sql-migrate"
)

//go:embed sql_migrations/*.sql
var dbMigrations embed.FS
var DbConnection *sql.DB

func DBMigrate(dbParam *sql.DB) {
    migrations := &migrate.EmbedFileSystemMigrationSource{
        FileSystem: dbMigrations,
        Root:      "sql_migrations",
    }

    n, errs := migrate.Exec(dbParam, "postgres", migrations,
migrate.Up)

    if errs != nil {
        panic(errs)
    }

    DbConnection = dbParam

    fmt.Println("Migration success applied", n, migrations)
}
```

Kode program di atas digunakan untuk proses migrasi golang ke *database*. Baris kode program `//go:embed sql_migrations/*.sql` bukanlah sebuah komentar, melainkan baris tersebut berfungsi sebagai kode yang akan menyematkan seluruh file yang berekstensi `.sql` yang ada pada *folder* `sql_migrations` ke dalam variabel `dbMigrations`. Oleh karena itu, perintah ini **wajib** dituliskan sebelum nantinya membangun fungsi migrasi *database*. Pada bagian awal kode program, terdapat 2 pendefinisian variabel, yakni **dbMigrations** yang akan menyimpan hasil embed yang telah dilakukan pada *folder* `sql_migrations` dan **dbConnection** yang akan menyimpan koneksi ke *database*.

Berikutnya terdapat *function* **DBMigrate()** yang di dalamnya terdapat parameter `dbParam` yang berfungsi dalam menerima status koneksi golang ke *database*. Ketika *function* tersebut dipanggil, maka sistem akan menjalankan proses migrasi *database* dengan root yang diambil dari *folder* `sql_migrations`. Berikutnya sistem akan menjalankan proses migrasi dengan pemanggilan terhadap fungsi **Exec()**. Proses tersebut akan menyimpan jumlah migrasi yang berhasil dilakukan dan mengembalikan kondisi error jika proses migrasi mengalami permasalahan. Jika terjadi error, maka sistem akan memanggil fungsi **panic()** yang akan langsung menghentikan jalannya program. Jika tidak terdeteksi error, maka sistem akan menampilkan pesan bahwa proses migrasi berhasil dilakukan.

Kemudian bukalah file **servo\_repo.go** dan masukkanlah kode program berikut:

```
package repositories

import (
    "database/sql"
    "mcs_bab_7/entities"
)

func InitProj(db *sql.DB) (err error) {
    sql := "INSERT INTO status(id, srv_status) values(1, 0)"
    _, err = db.Query(sql)
    return err
}
```



```

func GetStatus(db *sql.DB) (result []entities.Servo, err error) {
    sql := "SELECT * FROM status"
    rows, err := db.Query(sql)

    if err != nil {
        return
    }

    defer rows.Close()

    for rows.Next() {
        var data entities.Servo
        err = rows.Scan(&data.Id, &data.SrvStatus)
        if err != nil {
            return
        }
        result = append(result, data)
    }
    return
}

func UpdateStatus(db *sql.DB, status entities.Servo) (err error)
{
    sql := "UPDATE status SET srv_status = $1 WHERE id = 1"
    _, err = db.Exec(sql, status.SrvStatus)
    return
}

```

Kode di atas digunakan agar golang dapat melakukan interaksi dengan *database*. Terdapat 3 fungsi yang dibentuk pada file ini, antara lain **InitProj()**, **GetStatus()** dan **UpdateStatus()** yang masing-masing *function* memiliki tujuan penggunaannya sendiri. Fungsi InitProj() digunakan untuk menginisialisasi data awal dari status servo. Fungsi ini akan memberikan nilai 1 untuk id dan nilai 0 untuk srv\_status. Fungsi ini hanya dapat dilakukan 1x saja, karena *query* id yang didefinisikan adalah 1, sedangkan id sendiri merupakan *primary key* yang apabila

dijalankan kembali, maka akan terjadi kesalahan. Fungsi `GetStatus()` digunakan untuk membaca seluruh data yang tersimpan pada tabel status dengan menggunakan perintah *query* **SELECT \* FROM card**. Fungsi `UpdateStatus()` digunakan untuk merubah nilai pada `srv_status` dengan menggunakan perintah *query* **UPDATE status SET srv\_status = \$1 WHERE id = 1**. Selanjutnya masuklah ke dalam file **servo\_controller.go** dan masukkanlah kode program berikut:

```
package controllers

import (
    "mcs_bab_7/database"
    "mcs_bab_7/entities"
    "mcs_bab_7/repositories"
    "net/http"
    "strconv"

    "github.com/gin-gonic/gin"
)

func InitProj(c *gin.Context) {
    err := repositories.InitProj(database.DbConnection)

    if err != nil {
        c.JSON(http.StatusInternalServerError,
            gin.H{"error": err.Error()})
    }

    c.JSON(http.StatusOK, gin.H{})
}

func GetStatus(c *gin.Context) {
    var result gin.H
    status, err := repositories.GetStatus(database.DbConnection)

    if err != nil {
        result = gin.H{
```

```

        "result": err.Error(),
    }
} else {
    result = gin.H{
        "result": status,
    }
}

c.JSON(http.StatusOK, result)
}

func UpdateStatus(c *gin.Context) {
    var status entities.Servo
    srv_status, _ := strconv.Atoi(c.Param("srv_status"))
    status.SrvStatus = srv_status
    err := repositories.UpdateStatus(database.DbConnection,
status)

    if err != nil {
        c.JSON(http.StatusInternalServerError,
gin.H{"error": err.Error()})
        return
    }

    c.JSON(http.StatusOK, gin.H{"srvStatus":
status.SrvStatus})
}

```

Kode program yang digunakan pada file *controller* bertujuan untuk mengontrol apa yang akan dilakukan oleh sistem. Pada file ini, terdapat 3 fungsi yang dibentuk berdasarkan fungsi yang terbentuk pada file *servo\_repo.go*. Fungsi **InitProj()** berfungsi untuk menginisialisasi *field* id agar memiliki nilai 1 dengan memanggil fungsi *InitProj()* yang berada di *package* *repositories*. Fungsi **GetStatus()** digunakan untuk membaca table status dari *database* dengan memanggil fungsi *GetStatus()* yang berada di *package* *repositories*. Fungsi

**UpdateStatus()** digunakan untuk mengubah *field* `srv_status` dengan mengambil nilai dari *parameter* `srv_status`.

Selanjutnya, bukalah file **servo\_router.go** dan masukkanlah kode program berikut:

```
package routers

import (
    "mcs_bab_7/controllers"

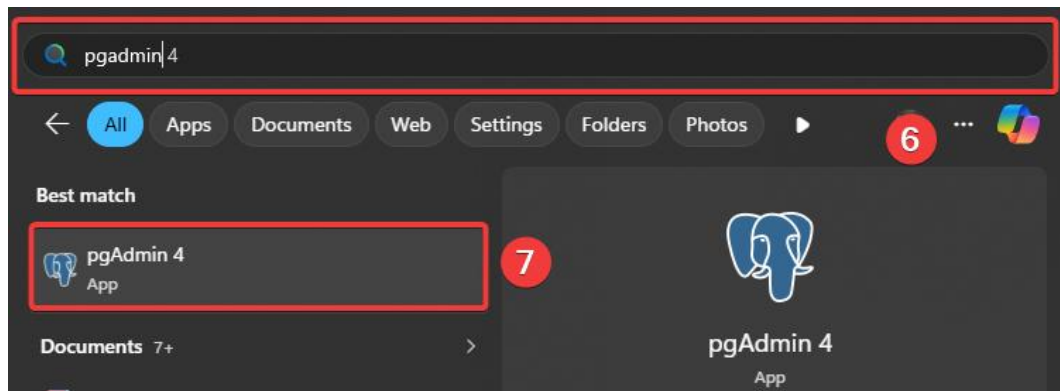
    "github.com/gin-gonic/gin"
)

func StartServer() *gin.Engine {
    router := gin.Default()
    router.POST("/servo/init-proj", controllers.InitProj)
    router.GET("/servo/status", controllers.GetStatus)
    router.PUT("/servo/update/:srv_status",
controllers.UpdateStatus)
    return router
}
```

Kode program yang dituliskan pada file tersebut merupakan kode yang akan mengatur *endpoint* dari masing-masing fungsi yang telah dibangun. Seluruh fungsi tersebut akan dijalankan dengan url yang sama. Namun, *endpoint* yang ingin digunakan akan disesuaikan berdasarkan kebutuhan. *Endpoint* yang dapat digunakan pada praktikum ini, antara lain:

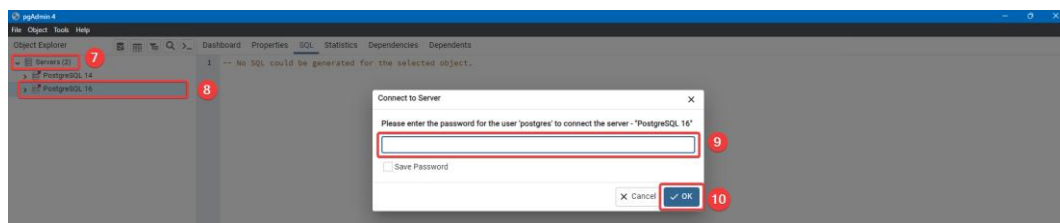
1. **/servo/init-proj** = Digunakan untuk menginisialisasi data awal dengan *method* API yang digunakan adalah *method* **POST**.
2. **/servo/status** = Digunakan untuk menampilkan seluruh data yang ada dengan menggunakan *method* **GET**.
3. **/servo/update/:srv\_status** = Digunakan untuk meng*update* data dari servo status dengan menggunakan *method* API **PUT**. Untuk meng*update* data, variabel `srv_status` pada *endpoint* diganti dengan data yang diinginkan

Setelah mendefinisikan router yang akan digunakan pada praktikum kali ini, langkah berikutnya sebelum membangun kode utama adalah membuat *database* terlebih dahulu. *Database* yang digunakan pada praktikum kali ini adalah postgres SQL yang dapat diakses dengan membuka *software* pgAdmin yang telah terinstall.



Gambar 7.6 Proses Membuka Postgre SQL

Setelah pgAdmin terbuka pada perangkat, tekanlah menu *server* yang berada pada bagian sebelah kiri dan pilihlah *server* PostgreSQL yang tersedia (**Note: Versi server dapat berbeda-beda**). Selanjutnya masukkanlah *password* yang telah dibuat ke dalam *field* yang telah disediakan dan tekanlah tombol OK untuk masuk ke dalam server tersebut.



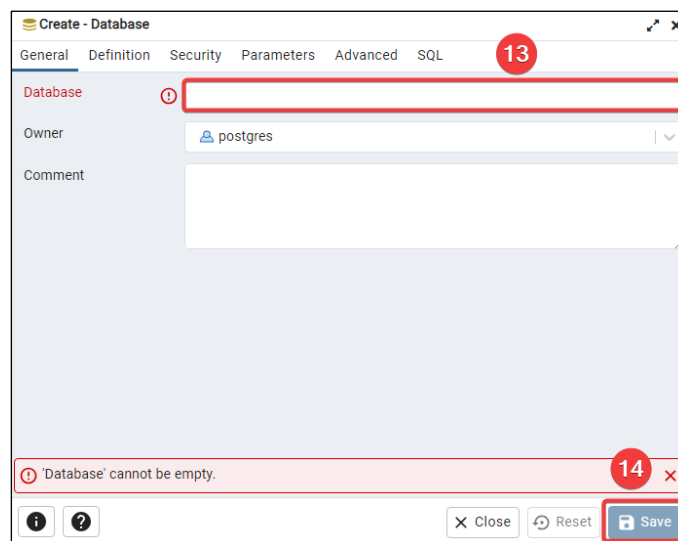
Gambar 7.7 Proses Mengakses Server Postgre SQL

Setelah berhasil masuk ke dalam server, klik kanan pada menu **Databases** > **Create** > **Database...**



Gambar 7.8 Proses Pembuatan Database

PostgreSQL akan menampilkan halaman baru yang berisikan konfigurasi untuk pembuatan *database*. Pada menu tersebut, isilah kolom **database** dengan nama bebas. Jika nama folder lebih dari 1 suku kata, pisahkan dengan menggunakan *underscore* (\_). Kemudian tekanlah tombol **save** untuk membuat *database*. Jika berhasil terbentuk, maka pada menu Databases yang ada di sebelah kiri, akan muncul file *database* dengan nama yang telah dibuat.



Gambar 7.9 Konfigurasi *Database*

Jika *database* telah terbentuk, kembalilah ke dalam *software visual studio code* dan masukkan kode berikut ke dalam file **main.go**

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "mcs_bab_7/database"
    "mcs_bab_7/routers"

    _ "github.com/lib/pq"
)
```

```

const (
    host      = "localhost"
    port      = 5432
    user      = "postgres"
    password  = "" // SESUAIKAN DENGAN PASSWORD
    POSTGRE YANG TELAH DIDAFTARKAN
    dbName    = "praktikum_mcs_bab_7" // SESUAIKAN DENGAN NAMA
    DATABASE YANG DIBUAT
)

var (
    DB  *sql.DB
    err error
)

func main() {
    var PORT = ":8080"

    psqlInfo := fmt.Sprintf(
        `host=%s port=%d user=%s password=%s dbname=%s
        sslmode=disable`,
        host, port, user, password, dbName,
    )

    DB, err = sql.Open("postgres", psqlInfo)

    if err != nil {
        log.Fatal("Error open DB", psqlInfo)
    }

    database.DBMigrate(DB)

    defer DB.Close()

    routers.StartServer().Run(PORT)
    fmt.Println("DB Success Connected")
}

```

Pada file tersebut, definisikan beberapa variabel yang bersifat konstanta, seperti **host**, **port**, **user**, **password**, dan **dbName**. Variabel tersebut ini nantinya akan digunakan untuk berkomunikasi dengan PostgreSQL. Selain itu, buatlah variabel global bernama **DB** dengan tipe `*sql.DB` dan **err** yang akan menangkap error.

Pada file tersebut, buatlah satu fungsi bernama **main()** yang di dalamnya terdapat logika program utama yang akan dijalankan oleh sistem. Pada fungsi tersebut definisikanlah variabel **PORT** dengan nilai `:8080`. SQL akan dibuka dengan pemanggilan terhadap fungsi **Open()** yang di dalamnya terdapat parameter **“postgres”** dan **psqlInfo**. Jika terjadi error pada saat membuka *database*, maka aplikasi akan menampilkan pesan error pada terminal. Selanjutnya, untuk proses migrasi *database*, panggilah fungsi **DBMigrate()** yang telah didefinisikan pada file *database* untuk menjalankan migrasi konfigurasi SQL ke aplikasi PostgreSQL. Kemudian, koneksi ke *database* akan ditutup setelah fungsi **main()** dijalankan dengan pemanggilan terhadap fungsi **Close()** dengan menggunakan `defer` agar tidak terjadi kebocoran koneksi. Kemudian, server akan mulai dijalankan dengan pemanggilan terhadap fungsi **StartServer()** yang telah didefinisikan pada file *routers* dan dijalankan pada port yang telah ditentukan.

Setelah kode pada *main.go* selesai dituliskan, bukalah terminal *visual studio code* dan ketikkan perintah **go run main.go** untuk menjalankan kode yang telah dibangun. Jika kode berhasil dijalankan, maka tampilan dari terminal akan terlihat, seperti pada Gambar 7.10.

```
$ go run main.go
Migration success applied 1 &{{0x7ff77f99ee60} sql_migrations}
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.

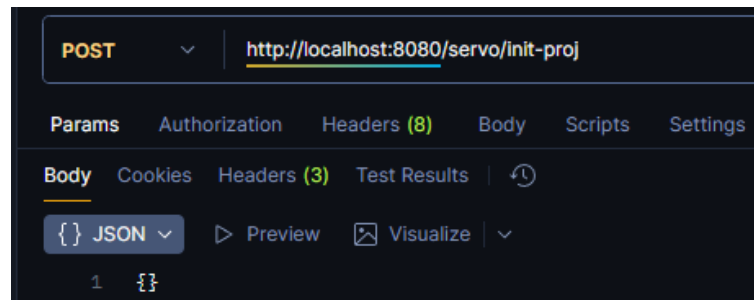
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
- using env:   export GIN_MODE=release
- using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] POST   /servo/init-proj      --> mcs_bab_7/controllers.InitProj (3 handlers)
[GIN-debug] GET    /servo/status         --> mcs_bab_7/controllers.GetStatus (3 handlers)
[GIN-debug] PUT    /servo/update/:srv_status --> mcs_bab_7/controllers.UpdateStatus (3 handlers)
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://github.com/gin-gonic/gin/blob/master/docs/doc.md#dont-trust-all-proxies for details.
[GIN-debug] Listening and serving HTTP on :8080
```

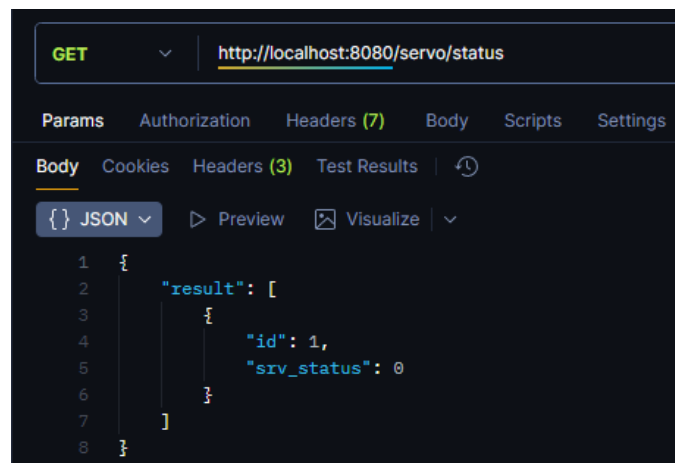
Gambar 7.10 Tampilan Terminal Ketika Berhasil Menjalankan *Backend*

Bukalah aplikasi *postman* pada perangkat dan lakukanlah uji coba terhadap beberapa *endpoint* yang telah dibangun.

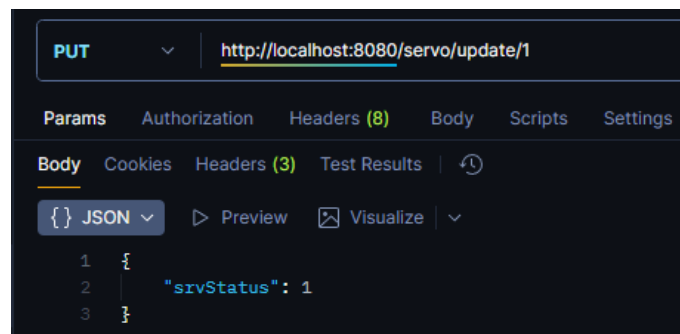




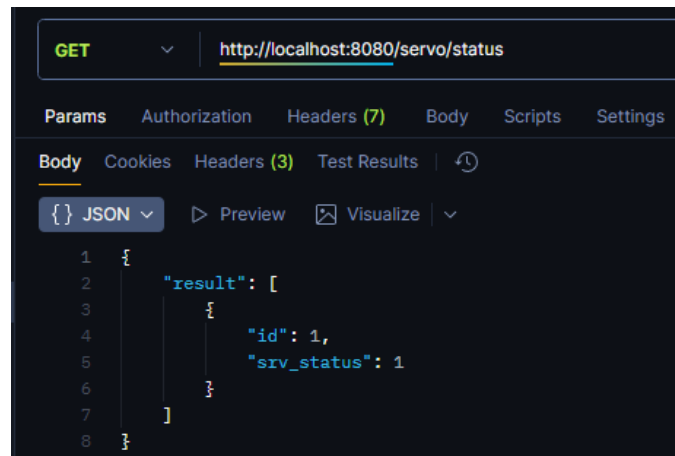
Gambar 7.11 Hasil Uji Coba Terhadap *Method* POST



Gambar 7.12 Hasil Uji Coba Terhadap *Method* GET



Gambar 7.13 Hasil Uji Coba Terhadap *Method* PUT



Gambar 7.14 Hasil Setelah Mengupdate Data Servo Status

```

$ go run main.go
Migration success applied 0 &{{0x7ff62db1ee60} sql_migrations}
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
- using env:   export GIN_MODE=release
- using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] POST   /servo/init-proj      --> mcs_bab_7/controllers.InitProj (3 handlers)
[GIN-debug] GET    /servo/status         --> mcs_bab_7/controllers.GetStatus (3 handlers)
[GIN-debug] PUT    /servo/update/:srv_status --> mcs_bab_7/controllers.UpdateStatus (3 handlers)
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://github.com/gin-gonic/gin/blob/master/docs/doc.md#dont-trust-all-proxies for details.
[GIN-debug] Listening and serving HTTP on :8080
[GIN] 2025/09/24 - 21:59:40 | 200 |      8.7518ms |      :::1 | POST | "/servo/init-proj"
[GIN] 2025/09/24 - 22:00:50 | 200 |     56.0903ms |      :::1 | GET  | "/servo/status"
[GIN] 2025/09/24 - 22:01:17 | 200 |      9.9558ms |      :::1 | PUT  | "/servo/update/1"
[GIN] 2025/09/24 - 22:01:40 | 200 |       547.2µs |      :::1 | GET  | "/servo/status"

```

Gambar 7.15 Tampilan Terminal Ketika Telah Menjalankan Beberapa *Method*