# BAB VII

# *SERVO CONTROLLER* (*BACKEND*)

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

## Tujuan Praktikum

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

## 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.

|  |  |
| --- | --- |
| *HARDWARE* YANG DIBUTUHKAN PRAKTIKUM | 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* |

## 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.

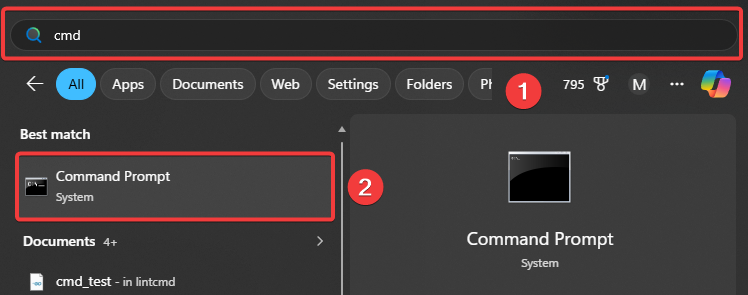
## Prosedur Praktikum

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:

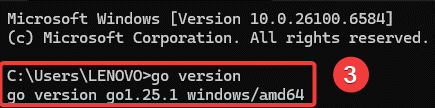
1. Memeriksa seluruh kebutuhan yang diperlukan

* **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 ter*install* pada perangkat, bukalah ***command prompt*** dan ketikan perintah **go version**. Jika perangkat telah ter*install* dengan bahasa golang, maka tampilan dari *command prompt* akan terlihat, seperti pada Gambar 7.1.

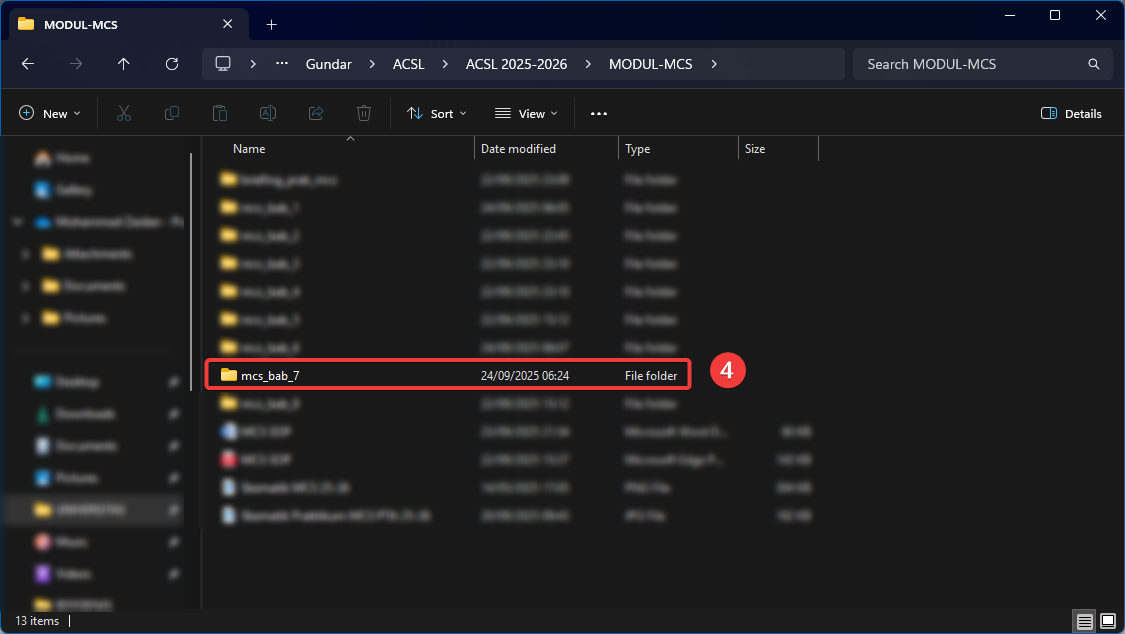


Gambar 7. Proses Pengecekkan Golang



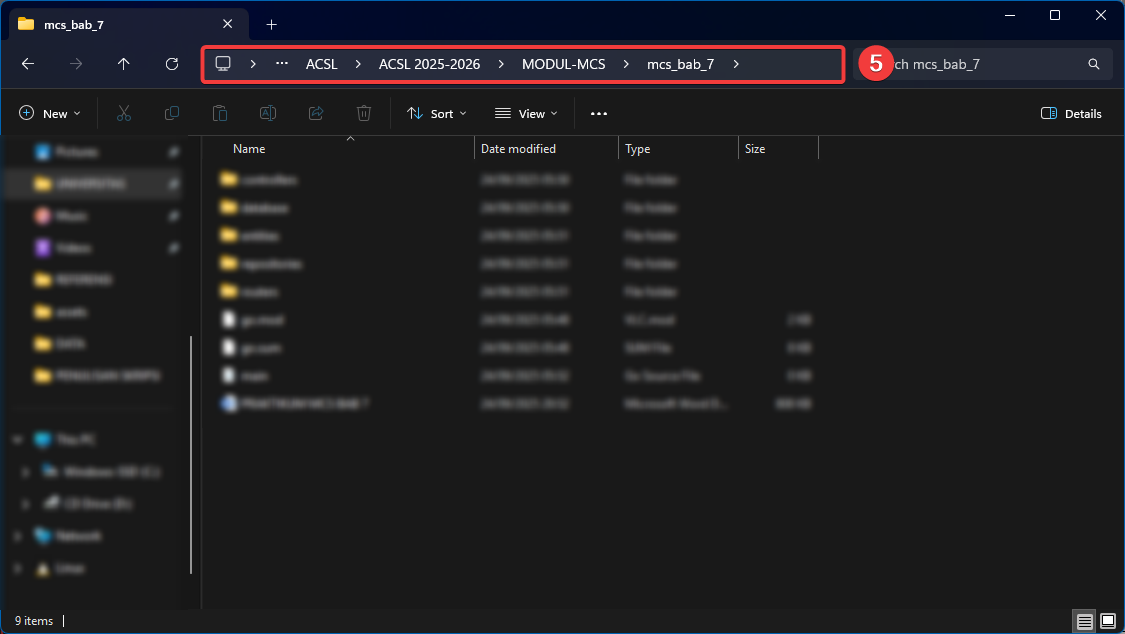
Gambar 7. Hasil Pengecekkan Versi Golang

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



Gambar 7. Proses Pembuatan *Folder Project*

1. 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. Proses Konfigurasi *Project*

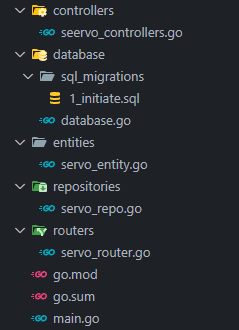
1. Setelah masuk ke dalam *command prompt,* masukkan seluruh konfigurasi berikut secara satu per satu.

|  |
| --- |
| 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 konfiurasi.

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. 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 *databse*.

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\_migrationsdan **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 akn 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.

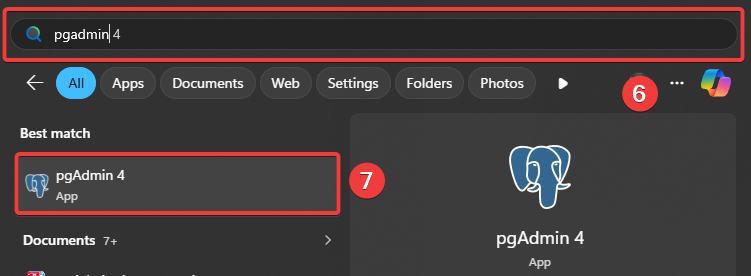
Selanjutnya, bukalah file **servo\_router.go** dan masukkanlah kode program berikut:

|  |
| --- |
| 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:

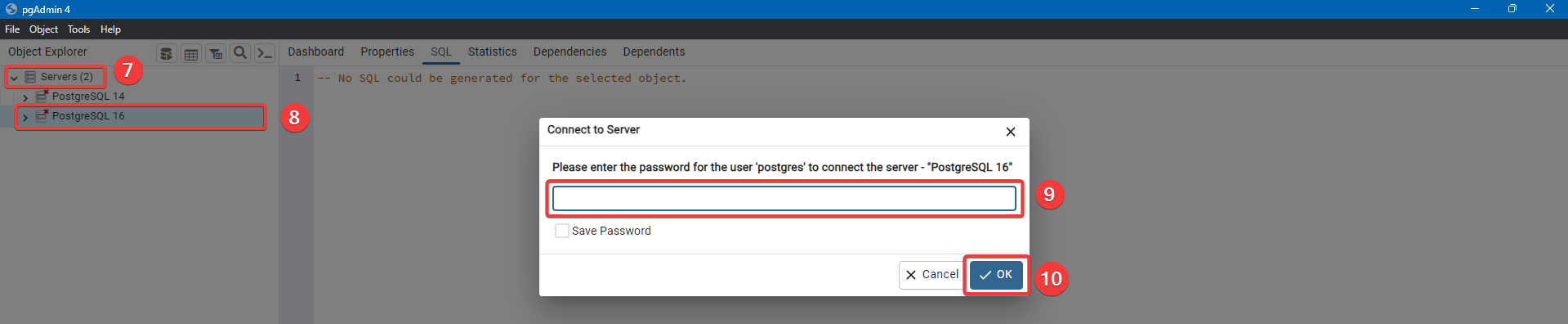
1. **/servo/init-proj** = Digunakan untuk menginisialisasi data awal dengan *method* API yang digunakan adalah *method* **POST**.
2. **/servo/status** = Digunakan untuk menampilkan seluruh data yang ada dengan menggunakan methodm **GET**.
3. **/servo/update/:srv\_status** = Digunakan untuk meng*update* 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 postgre SQL yang dapat diakses dengan membuka *software* pgAdmin yang telah ter*install.*



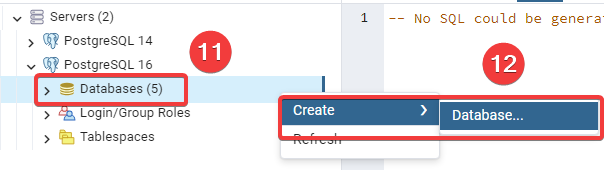
Gambar 7. Proses Membuka Postgre SQL

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.



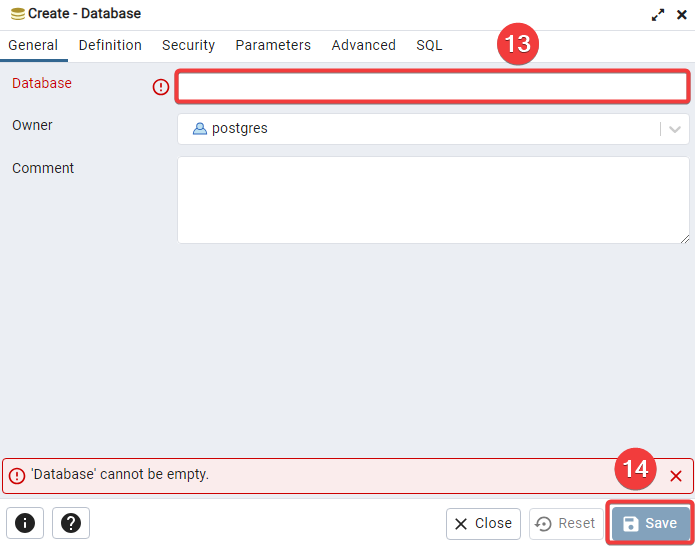
Gambar 7. Proses Mengakses Server Postgre SQL

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



Gambar 7. 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. Konfigurasi *Database*

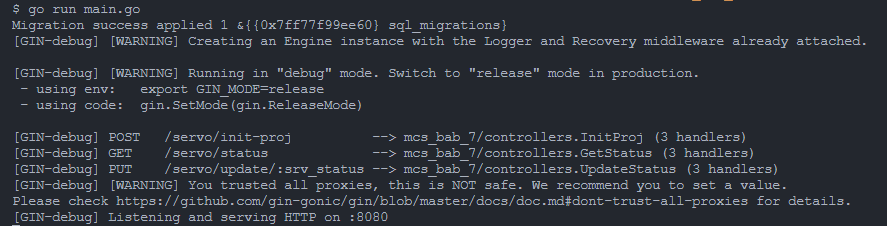
Jika *database* telah terbentuk, kembalilah ke dalam *software visual studio code* dan masukkan kode berikut ke dalam file **main.go**

|  |
| --- |
| 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.

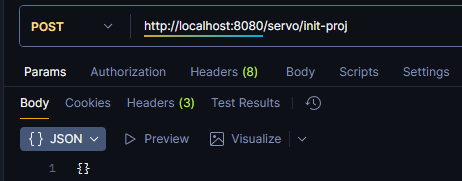
Pada file tersebut, buatlah satu fungsi bernama **main()** yang di dalamnya terdapat logika program utama yang akan dijalankan oleh sistem. Pada fungsi tersebut definisikanlah variabel PORT dengan nilai :8080. SQL akan dibuka dengan pemanggilan terhadap fungsi **Open()** yang di dalamnya terdapat parameter **“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 terjai 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 ketikan 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.

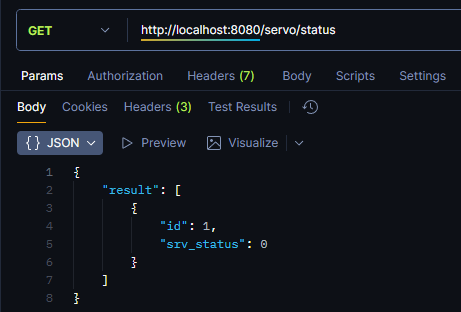


Gambar 7.TampilanTerminalKetika Berhasil Menjalankan *Backend*

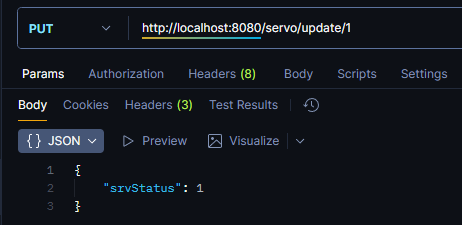
Bukalah aplikasi *postman* pada perangkat dan lakukanlah uji coba terhadap beberapa *endpoint* yang telah dibangun.



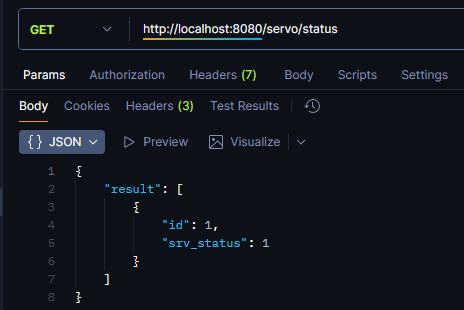
Gambar 7. Hasil Uji Coba Terhadap *Method* POST



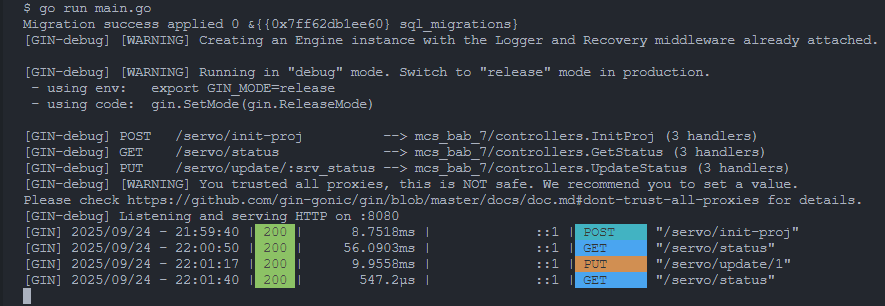
Gambar 7. Hasil Uji Coba Terhadap *Method* GET



Gambar 7. Hasil Uji Coba Terhadap *Method* PUT



Gambar 7. Hasil Setelah Meng*update* Data Servo Status



Gambar 7. Tampilan Terminal Ketika Telah Menjalankan Beberapa *Method*