BAB VI
CARD BRIDGE (*BACKEND* - GOLANG)

Pada praktikum MCS bab 6, praktikan akan belajar tentang bagaimana caranya membangun RESTFUL API sebuah aplikasi *backend* yang menjadi penghubung antara *end user* dengan *Internet of Things* (IoT). Server menjadi *connector* antara aplikasi android *mobile* dan IOT yang dibangun berbasis *Uniform Resource Locator* (URL). Pada praktikum kali ini, server akan berperan sebagai *database* untuk menyimpan nilai id yang dihasilkan dari RFID.

6.1 Tujuan Praktikum

Tujuan	Penjelasan
Memberikan penjelasan tentang bagaimana pengguna terhubung ke server.	Memberikan gambaran umum mengenai alur saat *user* melakukan koneksi ke dalam server
Membangun *database* dengan bahasa pemrograman Golang	Dalam bab ini, praktikan akan diajarkan cara membuat sebuah *database* dengan menggunakan bahasa pemrograman Golang
Melakukan migrasi *database* dengan Golang	Dalam bab ini, praktikan akan diajarkan bagaimana caranya melakukan migrasi ke Postgre SQL *database* menggunakan bahasa Golang

6.2 Persyaratan Praktikum

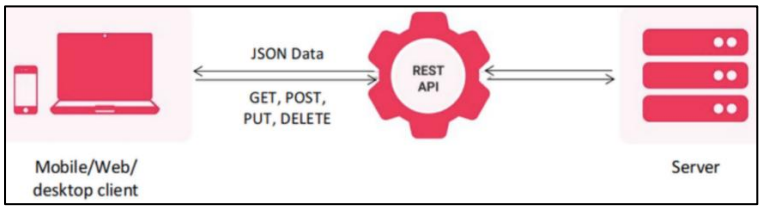
Disarankan praktikan menggunakan *hardware* dan *software* sesuai pada dokumentasi ini. Apabila terdapat versi yang lumayan lampau dari versi yang direkomendasikan atau *hardware* yang lawas maka sebaiknya bertanya kepada Asisten Mengajar Shift.

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

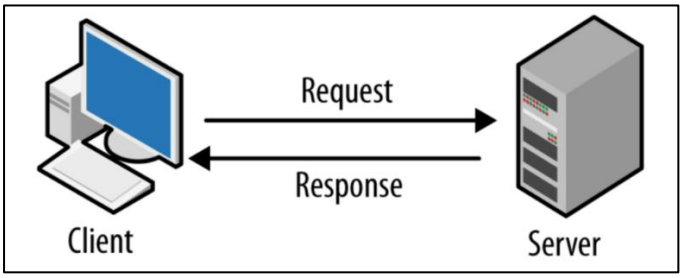
6.3 Materi Praktikum

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 *sensor Radio Frequency Identification* (RFID). Setiap kartu yang dibaca oleh RFID akan masuk ke *server* kemudian data kartu yang ada di *server* akan dibaca oleh aplikasi Android. Untuk berkomunikasi dengan *backend* berbasis URL (*RESTFUL* API) diperlukan beberapa *method*, beberapa *method* yang digunakan adalah **GET**, **POST**, **PUT**, **DELETE**.



Gambar 6.1 Hubungan *User* dan *Server*

User mengirim *request* dengan beberapa *method* yang digunakan untuk berkomunikasi dengan server kemudian server mengirim kembali data berupa *response*.



Gambar 6.2 *Request* dan *Response*

Di praktikum ini hanya akan membangun dari sisi server saja dan praktikum akan berlanjut di bab 7 dan 8

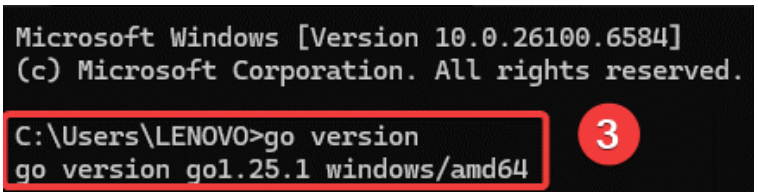
6.4 Prosedur Praktikum

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**

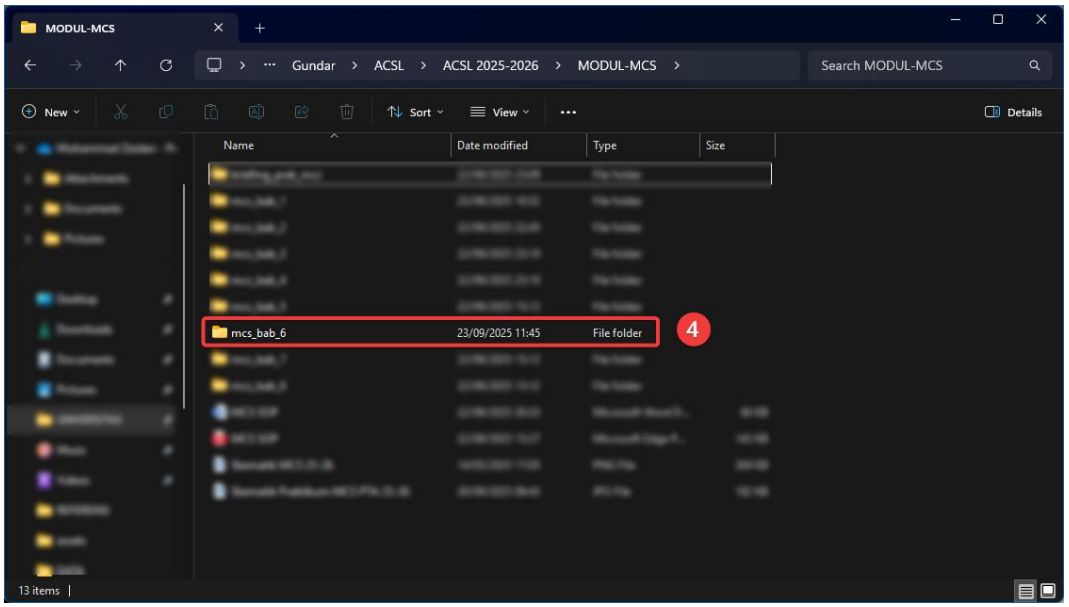
Untuk memastikan apakah bahasa pemrograman golang telah ter*install* pada perangkat, bukalah ***command prompt*** dan ketikan perintah **go version**. Jika perangkat telah ter*install* dengan bahasa golang, maka tampilan dari *command prompt* akan terlihat, seperti pada Gambar 6.4.

Gambar 6.3 Proses Pengecekkan Golang



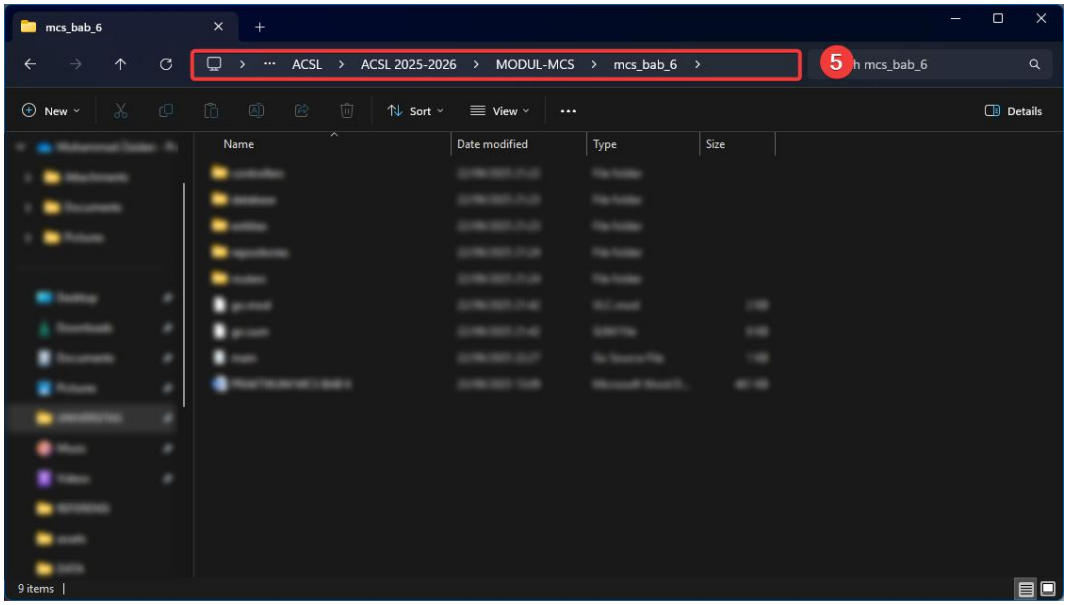
Gambar 6.4 Hasil Pengecekkan Versi Golang

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



Gambar 6.5 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 6.6 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"
```

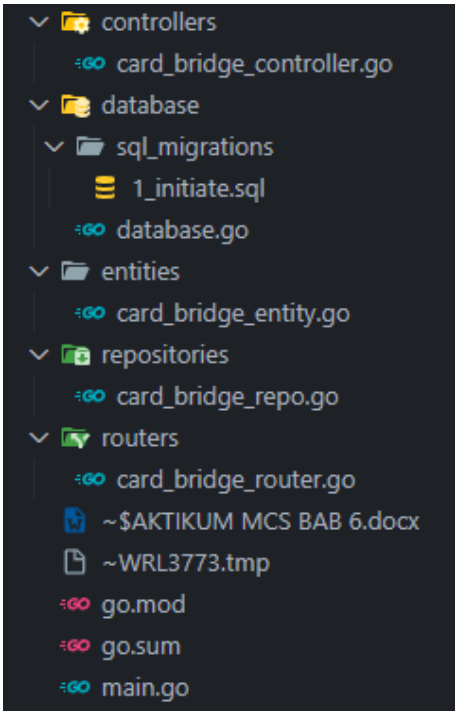
Berikut merupakan penjelasan singkat terkait kode konfigurasi yang telah dimasukkan:

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 konfiurasi.

Setelah melakukan konfigurasi pada *project* golang, bukalah *folder* tersebut pada *software visual studio code* dan bentuklah *tree project*, seperti yang terlihat pada Gambar 6.7.



Gambar 6.7 Struktur *Tree Project*

Setelah membentuk struktur *tree project*, bukalah file **card_bridge_entity.go** dan masukkanlah kode program berikut:

```
package entities

type Card struct {
	ID string `json:"id"`
}
```

Kode program tersebut berperan sebagai model yang mendefinisikan variabel serta tipe data yang digunakannya. Karena pada praktikum kali ini hanya digunakan untuk menyimpan data id dari RFID, maka variabel yang didefinisikan hanyalah 1 variabel saja dengan tipe string. Ketika data tersebut akan dikonversi ke dalam bentuk JSON, maka *field* tersebut akan tersimpan ke dalam *key* **id**.

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

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

CREATE TABLE card(
    id varchar(20)
);

-- +migrate StatementEnd
```

Kode program di atas digunakan untuk membuat tabel *database* baru bernama **card**. Tabel yang dibuat pada praktikum kali ini hanya memiliki 1 *field* bernama **id** bertipe data varchar yang hanya dapat menampung karakter sepanjang 20 karakter. **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("Migrations 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 **card_bridge_repo.go** dan masukkanlah kode program berikut:

```
package repositories

import (
	"database/sql"
	"mcs_bab_6/entities"
)

func GetCards(db *sql.DB) (result []entities.Card, err error) {
	sql := "SELECT * FROM card"
	rows, err := db.Query(sql)

	if err != nil {
		return
	}

	defer rows.Close()

	for rows.Next() {
```

```
            var data entities.Card
            err = rows.Scan(&data.ID)
            if err != nil {
                  return
            }
            result = append(result, data)
      }
      return
}

func InsertCard(db *sql.DB, card entities.Card) (err error) {
      sql := "INSERT INTO card(id) values($1)"
      _, err = db.Exec(sql, card.ID)
      return err
}

func DeleteCard(db *sql.DB, card entities.Card) (err error) {
      sql := "DELETE FROM card WHERE id = $1"
      _, err = db.Exec(sql, card.ID)
      return err
}
```

Kode di atas digunakan agar golang dapat melakukan interaksi dengan *database*. Terdapat 3 fungsi yang dibentuk pada file ini, antara lain **GetCards(), InsertCard()** dan **DeleteCard()** yang masing-masing *function* memiliki tujuan penggunaannya sendiri. Fungsi GetCards() digunakan untuk membaca seluruh data yang tersimpan dalam tabel card. Data tersebut dibaca dengan menggunakan perintah *query* **SELECT * FROM card**. Pada fungsi tersebut, sistem akan melakukan *looping* untuk mengisikan data ke dalam variabek result.

Fungsi InsertCard() digunakan untuk meng*input* data ke dalam table card dengan menggunakan perintah *query* **INSERT INTO card(id) values($1)**. Sedangkan, fungsi DeleteCard() merupakan fungsi untuk menghapus data dari tabel berdasarkan id yang terdeteksi. Perintah *query* yang digunakan untuk menghapus data tersebut adalah **DELETE FROM card WHERE id = $1**. Selanjutnya

masuklah ke dalam file **card_bridge_controller.go** dan masukkanlah kode program berikut:

```
package controllers

import (
	"mcs_bab_6/database"
	"mcs_bab_6/entities"
	"mcs_bab_6/repositories"
	"net/http"
	"github.com/gin-gonic/gin"
)

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

	if err != nil {
		result = gin.H{
			"result": err.Error(),
		}
	} else {
		result = gin.H{
			"result": card,
		}
	}

	c.JSON(http.StatusOK, result)
}

func InsertCard(c *gin.Context) {
	var card entities.Card
	idCard := c.Param("id")
	card.ID = idCard

	err := repositories.InsertCard(database.DbConnection, card)

	if err != nil {
```

```
            c.JSON(http.StatusInternalServerError,
gin.H{"errpr": err.Error()})
            return
      }

      c.JSON(http.StatusOK, card)
}

func DeleteCard(c *gin.Context) {
      var card entities.Card
      idCard := c.Param("id")
      card.ID = idCard

      err := repositories.DeleteCard(database.DbConnection, card)

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

      c.JSON(http.StatusOK,   gin.H{"message":   "Data   berhasil
dihapus", "id": idCard})
}
```

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 card_bridge_repo.go. Fungsi **GetCards()** berfungsi untuk mengambil seluruh data yang tersimpan pada *database*. Data yang diambil akan ditampung ke dalam forat JSON dengan *key* bernama result. Jika terjadi error, maka sistem akan menampilkan pesan error pada *key* tersebut.

Fungsi **InsertCard()** digunakan untuk meng*input* data ke dalam *database*. Data kartu dimasukkan menggunakan parameter di akhir url bernama id. Variabel idCard memiliki nilai yang diambil dari parameter id, lalu card.ID diinisialisasi dengan nilai dari idCard. Setelah itu memanggil InsertCard() dari *package*

repositories agar data masuk ke *database*. Jika error maka *response* yang akan diberikan adalah error dengan StatusInternalServerError. Fungsi **DeleteCard()** digunakan untuk meenghapus data kartu yang telah disimpan pada *database* dengan memanggil fungsi DeleteCard() yang telah dibentuk pada *package* repositories. Jika terjadi *error* maka response yang diberikan adalah StatusInternalServerError.

Selanjutnya, bukalah file **card_bridge_router.go** dan masukkanlah kode program berikut:

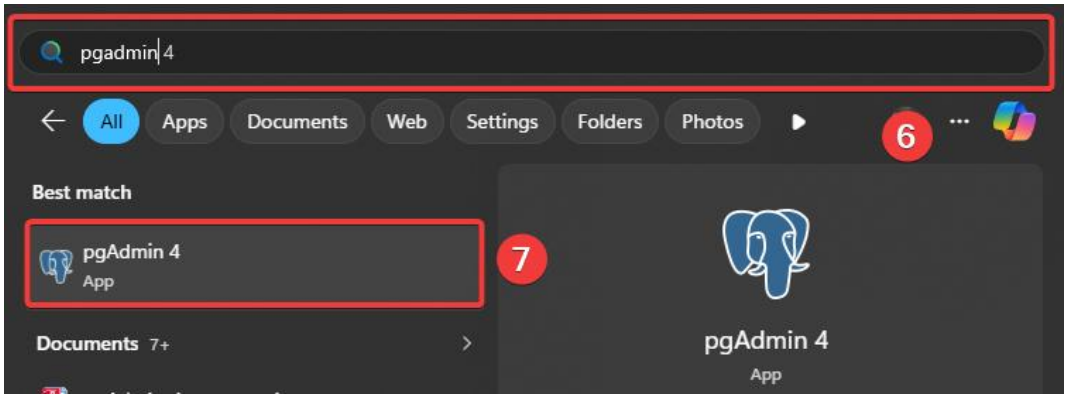
```
package routers

import (
	"mcs_bab_6/controllers"
	"github.com/gin-gonic/gin"
)

func StartServer() *gin.Engine {
	router := gin.Default()
	router.GET("/cards", controllers.GetCards)
	router.POST("/card/input/:id", controllers.InsertCard)
	router.DELETE("/card/delete/:id", controllers.DeleteCard)

	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. **/cards** = Digunakan untuk menampilkan seluruh data yang ada dengan *method* API yang digunakan adalah *method* **GET**.
2. **/card/input/:id** = Digunakan untuk meng*input* data baru ke dalam *database* dengan *method* API yang digunakan adalah *method* **POST**. Untuk menginput data, variabel id pada *endpoint* diganti dengan data yang diinginkan.

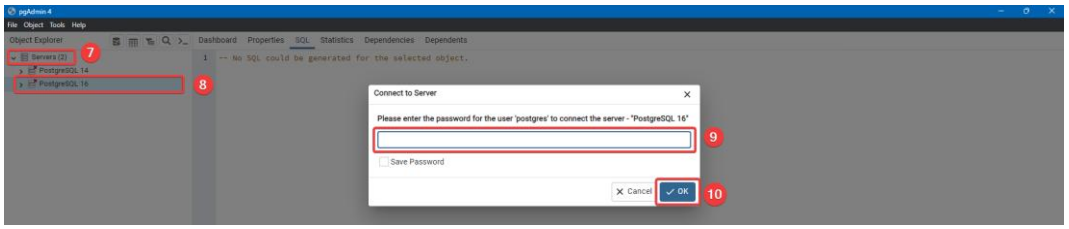
3. **/card/delete/:id** = Digunakan untuk menghapus data yang telah tersimpan dalam *database* dengan menggunakan *method* API **DELETE**. Sama halnya dengan pada saat *input* data, variabel id pada *endpoint* tersebut juga diganti dengan data yang ingin dihapus.

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.*



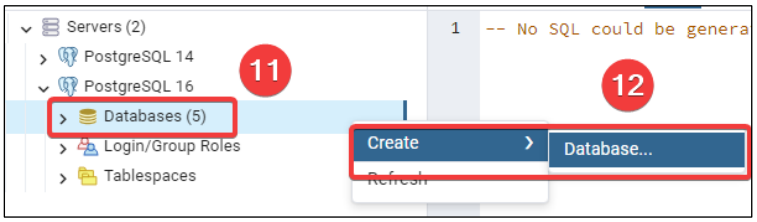
Gambar 6.8 Proses Membuka Postgre SQL

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.



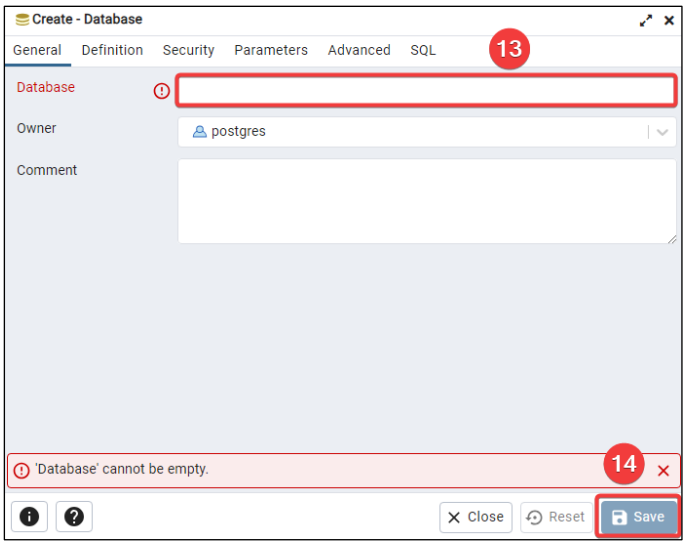
Gambar 6.9 Proses Mengakses Server Postgre SQL

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



Gambar 6.10 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 6.11 Konfigurasi *Database*

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

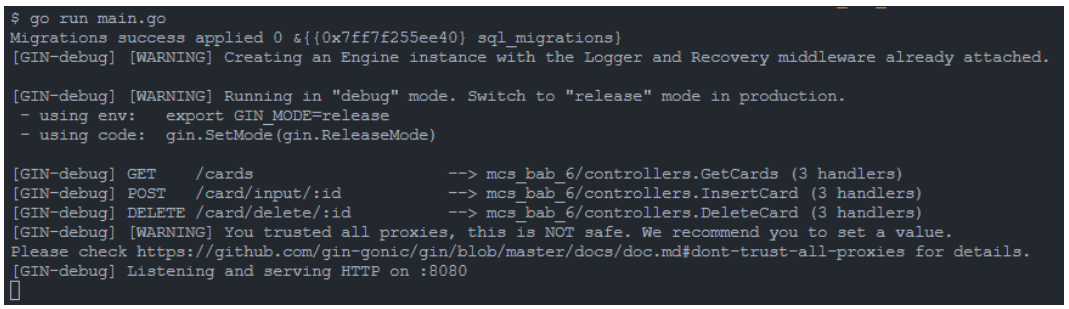
```
package main
import (
	"database/sql"
	"fmt"
	"log"
	"mcs_bab_6/database"
```

```
	"mcs_bab_6/routers"
	_ "github.com/lib/pq"
)

const (
	host     = "localhost"
	port     = 5432
	user     = "postgres"
	password = ""                    // SESUAIKAN DENGAN PASSWORD
POSTGRE YANG TELAH DIDAFTARKAN
	dbName   = "praktikum_mcs_bab_6" // 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.Fatalf("Error Open DB: %v\n", err)
	}

	database.DBMigrate(DB)
	defer DB.Close()
	routers.StartServer().Run(PORT)
	fmt.Printf("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.

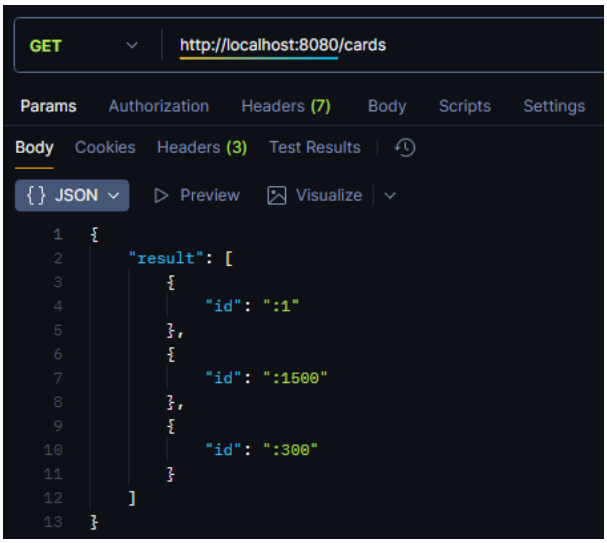
Setelah kode pada main.go selesai dituliskan, bukalah terminal *visual studio code* dan ketikan perintah **go run main.go** untuk menjalankan kode yang telah dibangun. Jika kode berhasil dijalankan, maka tampilan dari terminal akan terlihat, seperti pada Gambar 6.12.



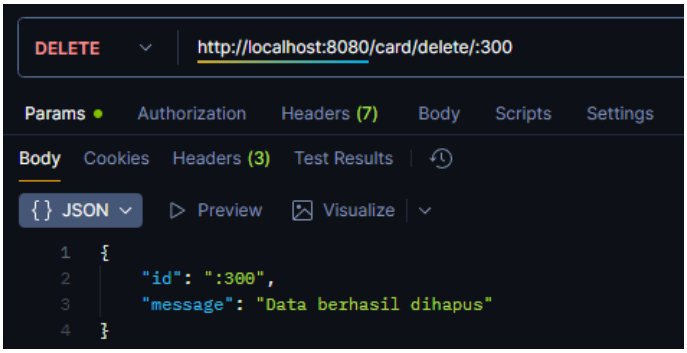
Gambar 6.12 Tampilan Terminal Ketika Berhasil Menjalankan *Backend*

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

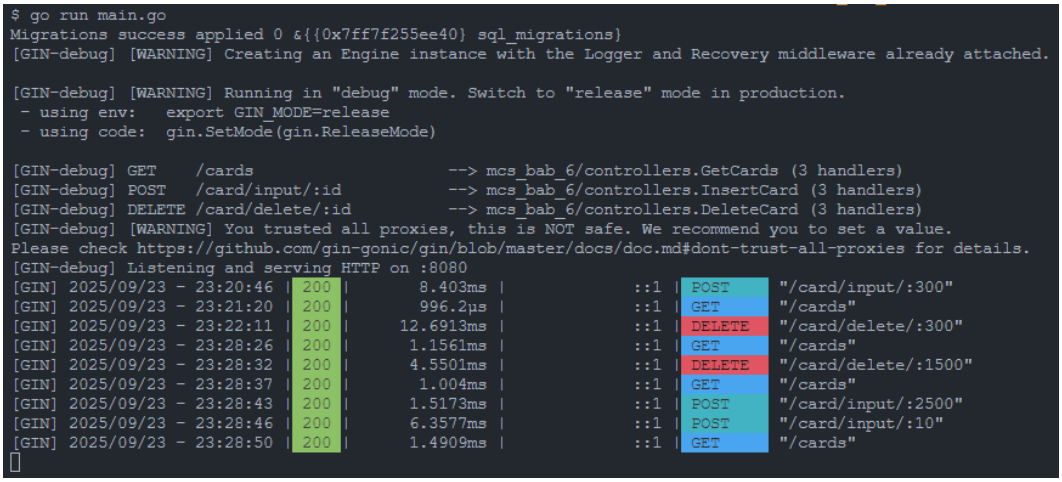
Gambar 6.13 Hasil Uji Coba Terhadap *Method* POST



Gambar 6.14 Hasil Uji Coba Terhadap *Method* GET



Gambar 6.15 Hasil Uji Coba Terhadap *Method* DELETE



Gambar 6.16 Tampilan Terminal Ketika Telah Menjalankan Beberapa *Method*