

# BAB 5

## Tahapan Pembuatan Back-end

---

Pada BAB ini akan membahas tentang perancangan bagian *Back-end* pada aplikasi ini. Hal ini bertujuan agar aplikasi mempunyai alur untuk bagian logikanya dengan cara membuat API. API bisa menghubungkan sistem internal dengan mudah, berkomunikasi antar platform, membuat API akan mempermudah pengembangan aplikasi karena akan melancarkan proses automasi dan pengembangan secara cepat (Iyengar, Khanna, Ramadath, & Stephens, 2017). Berikut adalah tahapan-tahapan dalam membuat API untuk E-Library.

### 5.1. Perancangan Endpoint Autentifikasi User

#### 1. Koneksi Database

```
"gorm.io/driver/postgres"  
"gorm.io/gorm"
```

*Gambar 1 Instalasi Gorm*

Hal yang pertama kali dilakukan adalah melakukan koneksi database dengan memasang *dependency* terlebih dahulu. Pemasangan *dependency* dapat dilakukan dengan cara mengikuti intruksi seperti pada Gambar 37 pada terminal di VSCode yang terpasang pada project, hasil dari pemasangan *dependency* bisa dilihat pada file yang bernama "go.mod".

```

42  gorm.io/driver/postgres v1.5.2 // indirect
43  gorm.io/gorm v1.25.1 // indirect
44  )
45

```

Gambar 2 Hasil Instalasi Gorm

Pada Gambar 38 merupakan keterangan Gorm sudah terpasang pada *dependency list* dalam file yang bernama “go.mod”. Gorm merupakan *library* yang mengimplementasikan teknik dan menyediakan layer yang berbasis *object-oriented* diantara *database* dan bahasa pemrograman OOP Go (Tomilov, 2020).

## 2. Membuat Initializer Environment

Tahapan selanjutnya adalah membuat initializer yang berfungsi untuk membuat koneksi antara API dan database. Hal tersebut dapat dilakukan dengan cara berikut.



Gambar 3 File .env

Buat file baru yang bernama .env yang digunakan untuk menampung *Environment Variable*. Hal tersebut berfungsi untuk menampung alamat dari database, JWT *Secret Key*, dan port API untuk dijalankan.

```

PORT=3000

SECRET=asd112j3ihshu192

DB="host=localhost user=postgres password=ahmad dbname=elibrary_app
port=5432 sslmode=disable"

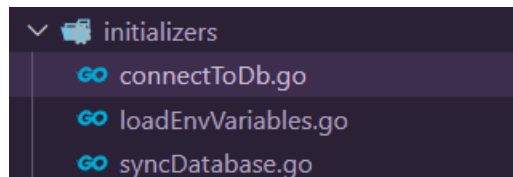
```

Kolom 1 Isi .env

Lalu masukkan *source code* yang ada pada Kolom 3 yang berisi *PORT*, *Secret Key*, dan *DB Address*. Isi dari port dapat disesuaikan dengan pengaturan pada device Anda.

### 3. Membuat File Initializer

Proses selanjutnya adalah membuat file dari *initializer* itu sendiri. Disini initializer terdapat tiga file seperti pada Gambar 40 yaitu “connectToDB.go”, “loadEnvVariables.go”, dan “syncDatabase.go”. File tersebut masing-masing mempunyai kegunaanya yaitu “connectToDB” berfungsi untuk membuat koneksi ke database, “loadEnvVariables” digunakan untuk memuat file .env supaya bisa terbaca oleh program, dan terakhir “syncDatabase” digunakan untuk migrasi model ke database agar tabel bisa dibuat secara otomatis.



Gambar 4 File Initializers

Selanjutnya adalah memasukkan *source* code dari setiap file dengan mengikuti kolom berikut.

```
package initializers

import (
    "os"

    "gorm.io/driver/postgres"
    "gorm.io/gorm"
)

var DB *gorm.DB
```

```
func ConnectToDb() {
    var err error
    dsn := os.Getenv("DB")
    DB, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})

    if err != nil {
        panic("failed to connect to database!")
    }
}
```

*Kolom 2 ConnectToDb Initializer*

Pada kolom 4 adalah source code untuk melakukan koneksi ke database. Pertama akan dilakukan import dahulu library untuk gorm dan gorm/postgres yang berfungsi untuk melakukan komunikasi antar database lalu ada function untuk menampung method dan variabel yang berfungsi untuk memanggil .env agar alamat dapat terdeteksi oleh gorm.

```
package initializers

import (
    "log"

    "github.com/joho/godotenv"
)

func LoadEnvVariables() {
    err := godotenv.Load()

    if err != nil {
        log.Fatal("error loading .env file")
    }
}
```

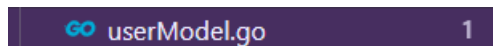
*Kolom 3 Source Code LoadEnvVariables*

Pada Kolom 5 merupakan *code* yang digunakan untuk mencari variabel yang terkandung dalam file .env. Namun sebelum melakukan akses terhadap variabel kita harus melakukan instalasi terlebih dahulu untuk mendeteksi file

.env yang sudah dibuat dengan cara mengetikkan “go install github.com/joho/godotenv”. Apabila sudah melakukan instalasi maka pembuatan *function* bisa dilakukan dan library “godotenv” akan dipanggil dalam function tersebut.

#### 4. Pembuatan Model

Apabila struktur dari initializer sudah dibuat, selanjutnya adalah membuat model dari User. Model ini akan digunakan untuk membuat tabel secara otomatis menggunakan *library* Gorm dari golang. Berikut adalah cara membuat model tersebut.



*Gambar 5 File User Model*

Buat folder yang bernama Models lalu buat file dengan nama “userModel.go”.

```
package models
import "gorm.io/gorm"
type User struct {
    gorm.Model
    Email    string `gorm:"unique"`
    Password string
    Name     string
}
```

*Kolom 4 Model User*

Pada Kolom 6 merupakan struktur dari model dari User, pertama akan dilakukan terlebih dahulu *import library* dari gorm yang digunakan membuat tabel. Selanjutnya ada properti dari model itu sendiri yang diberi nama “User”

yang berisi “gorm.Model” yang berguna untuk mencetak kolom id, date created, dan deleted secara otomatis oleh *library* gorm. Lalu ada Email, Password, dan Name yang berfungsi untuk mencetak kolom pada database.

```
package initializers

import "elab_v2/models"

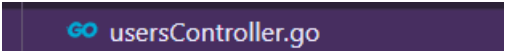
func SyncDatabase() {
    DB.AutoMigrate(&models.User{})
}
```

*Kolom 5 Source SyncDatabase*

Kembali ke folder initializers, disini akan dibuat file yang bernama “syncDatabase.go” yang digunakan untuk melakukan *automigrate*. *Automigrate* berfungsi untuk membuat tabel secara otomatis kedalam RDBMS berdasarkan model yang telah dibuat. Model User akan dipanggil dengan cara melakukan *import* terlebih dahulu *package* models, lalu model user akan dipanggil dalam function yang bernama *SyncDatabase* seperti pada Kolom 7.

## 5. Pembuatan Controller User

Setelah tahap pembuatan Initializer dan Model sudah dibuat maka tahapan selanjutnya adalah membuat controller yang berfungsi untuk membuat logika untuk endpoint yang akan dipanggil nantinya.



go usersController.go

*Gambar 6 File usersController*

Buatlah folder yang bernama models, lalu buat file yang bernama “usersController”.

```
package controllers

import (
    "elib_v2/initializers"
    "elib_v2/models"
    "net/http"
    "os"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/golang-jwt/jwt/v4"
    "golang.org/x/crypto/bcrypt"
)
```

#### *Kolom 6 Import Library*

Setelah membuat file selanjutnya adalah mengimport *library* seperti pada Kolom 8 yang akan digunakan untuk membuat function dari controller.

```
func Signup(c *gin.Context) {

    var body struct {
        Email    string
        Password string
        Name     string
    }

    if c.Bind(&body) != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to read body",
        })

        return
    }
}
```

#### *Kolom 7 Function Signup Pembuatan Property*

Lalu membuat variabel *body* yang nantinya akan dipanggil untuk membuat method dari pembuatan user.

```

//hash password disini
hash, err := bcrypt.GenerateFromPassword([]byte(body.Password),
10)

if err != nil {
    c.JSON(http.StatusBadRequest, gin.H{
        "error": "failed to hash password",
    })

    return
}

// membuat user
user := models.User{Email: body.Email, Password: string(hash),
Name: body.Name}
result := initializers.DB.Create(&user)

```

*Kolom 8 Hashing dan penampungan body*

Selanjutnya adalah pembuatan variabel untuk melakukan enkripsi pada password menggunakan bcrypt seperti pada kolom 10. Juga pembuatan variabel user untuk mendeklarasikan model dari user dan menampung *value* dari model user.

```

if result.Error != nil {
    c.JSON(http.StatusBadRequest, gin.H{
        "error": "Failed to create user",
    })

    return
}

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

```

*Kolom 9 Method error untuk signup*

Pada Kolom 11 adalah method untuk menunjukkan error apabila request yang dilakukan adalah salah. Apabila request dilakukan dengan benar maka akan direspon dengan respon StatusOK HTTP 200.



```
func Login(c *gin.Context) {
    var body struct {
        Email    string
        Password string
    }

    // failed read body
    if c.Bind(&body) != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": "Failed to read body",
        })

        return
    }
}
```

*Kolom 10 Function Login*

Selanjutnya setelah membuat function Signup untuk melakukan registrasi, maka tahapan selanjutnya adalah membuat function login. Pertama adalah membuat variabel *constructor* seperti pada Kolom 12. Selanjutnya membuat respon apabila ada error pada saat membaca body dengan menggunakan method if dan http StatusBadRequest.

```
var user models.User
initializers.DB.First(&user, "email = ?", body.Email)

if user.ID == 0 {
    c.JSON(http.StatusBadRequest, gin.H{
        "error": "invalid email or password",
    })

    return
}
```

*Kolom 11 Comparator Email*

Setelah membuat constructor selanjutnya adalah membuat comparator untuk kolom Email yang terdapat pada Kolom 13. Variabel user akan memanggil kolom tertentu dengan memanggil initializer, apabila email salah akan direspon

dengan output http *Bad Request*.

```
err := bcrypt.CompareHashAndPassword([]byte(user.Password),
[]byte(body.Password))

if err != nil {
    c.JSON(http.StatusBadRequest, gin.H{
        "error": "invalid email or password",
    })

    return
}

//generate token
token := jwt.NewWithClaims(jwt.SigningMethodHS256,
jwt.MapClaims{
    "foo": user.ID,
    "exp": time.Now().Add(time.Hour * 24 * 30).Unix(),
})

if err != nil {
    c.JSON(http.StatusBadRequest, gin.H{
        "error": "generate token gagal!",
    })

    return
}
```

*Kolom 12 Komparasi password*

Berikutnya adalah melakukan komparasi password yang telah melalui tahap enkripsi seperti pada Kolom 14. Pada variabel “err” akan dipanggil kolom Password yang sudah dienkripsi dan akan dikomparasikan salah atau benarnya. Apabila password salah akan direspon dengan *Bad Request*, sedangkan apabila password benar token akan ter-*generate* dengan batas waktu tertentu, serta apabila proses *generate* token gagal akan direspon dengan *Bad request* dengan pesan “generate token gagal”.

```

    c.SetSameSite(http.SameSiteLaxMode)
    c.SetCookie("Authorization", tokenString, 3600*24*30, "", "",
false, true)

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

```

### *Kolom 13 Penyimpanan Token*

Terakhir adalah tahap penyimpanan token pada Kolom 15. Token akan dicetak dalam bentuk string yaitu tokenString dan akan kadaluarsa dalam waktu tertentu. Token akan disimpan dalam cookie pada browser atau pada cookie management aplikasi pengujian API seperti Postman.

## 6. Pembuatan Middleware

Middleware adalah perangkat yang terdapat dalam aplikasi yang mempunyai peran kunci yaitu mengintegrasikan data dengan berbagai perangkat, memungkinkan kita untuk berkomunikasi antar perangkat, serta mengatur keputusan apabila aplikasi tersebut membutuhkan perizinan (Cruz, Rodriguez, Al-Muhtadi, Korotaev, & Alberquerque, 2018). Berikut adalah tahapan yang akan dilewati pada saat membuat middleware.

```

package middleware

import (
    "eliv_v2/initializers"
    "eliv_v2/models"
    "fmt"
    "net/http"
    "os"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/golang-jwt/jwt/v4"
)

```

### *Kolom 14 Import Untuk Middleware*

Pertama adalah lakukan import terlebih dahulu kebutuhan *library* untuk

Middleware. Import ini mencakup *package* dan *library* yang terdiri dari initializers, model, gin, jwt, dll.

```
func RequireAuth(c *gin.Context) {
    //get cookie
    tokenString, err := c.Cookie("Authorization")

    if err != nil {
        c.AbortWithStatus(http.StatusUnauthorized)
    }

    //validasi
    token, err := jwt.Parse(tokenString, func(token *jwt.Token)
(interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("Unexpected signing method: %v",
token.Header["alg"])
        }

        // hmacSampleSecret is a []byte containing your secret, e.g.
[]byte("my_secret_key")
        return []byte(os.Getenv("SECRET")), nil
    })

    if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
        //cek waktu expire
        if float64(time.Now().Unix()) > claims["exp"].(float64) {
            c.AbortWithStatus(http.StatusUnauthorized)
        }

        //mencari user dengan token
        var user models.User
        initializers.DB.First(&user, claims["sub"])

        if user.ID == 0 {
            c.AbortWithStatus(http.StatusUnauthorized)
        }

        //memasangkan dengan request
        c.Set("user", user)
        //Lanjut
        c.Next()
    } else {
        c.AbortWithStatus(http.StatusUnauthorized)
    }

    c.Next()
}
```

#### Kolom 15 Function RequireAuth

Function diatas digunakan untuk menampung method-method yang berhubungan dengan autentifikasi user. Pertama adalah variabel tokenString

yang digunakan untuk mengambil *cookie* yang di *generate* oleh *usersController*. Apabila tidak ditemukan *cookie* maka proses tidak akan dilanjutkan dan akan mengeluarkan output berupa HTTP Response yaitu *StatusUnauthorized*, sedangkan apabila token ditemukan di *cookie* maka akan dilanjutkan dengan melakukan set user sesuai dengan data dari tabel user tersebut.

## 7. Pembuatan URL Endpoint Autentifikasi

Proses terakhir dalam pembuatan *Endpoint* untuk autentifikasi pengguna adalah membuat URL dari *controller* yang sudah dibuat. Berikut adalah tahapannya.

```
package main

import (
    "elib_v2/controllers"
    "elib_v2/initializers"
    "elib_v2/middleware"

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

func init() {
    initializers.LoadEnvVariables()
    initializers.ConnectToDb()
    initializers.SyncDatabase()
}

func main() {
    r := gin.Default()

    api := r.Group("/api")
    {
        //auth
        api.POST("/signup", controllers.Signup)
        api.POST("/login", controllers.Login)
    }
    r.Run()
}
```

Kolom 16 *main.go*

Buka kembali file *main.go* yang berada dalam direktori utama API, lalu import semua dependency yang dibutuhkan seperti *package-package* yang

sudah dibuat seperti pada tahap sebelumnya. Setelah mengimport *library* yang dibutuhkan selanjutnya adalah membuat function untuk menampung initializer. Terakhir adalah membuat function untuk rute URL dengan cara membuat variabel “r” untuk rute dan “api” untuk menyatukan api dalam satu rute URL yaitu “/api”.

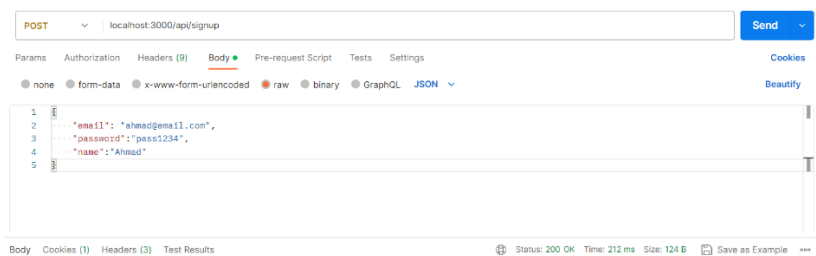
## 8. Uji Coba Endpoint User

Setelah semua tahapan dilalui selanjutnya adalah melakukan ujicoba terhadap Endpoint yang sudah dibuat. Pertama jalankan terlebih dahulu API dengan cara mengetikkan input “go run main.go” pada terminal dalam text editor atau terminal “cmd”.

```
2023/05/25 06:54:26 stdout: Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.  
2023/05/25 06:54:26 stdout: [GIN-debug] Environment variable PORT="3000"  
2023/05/25 06:54:26 stdout: [GIN-debug] Listening and serving HTTP on :3000  
2023/05/25 07:57:52 Running build command!
```

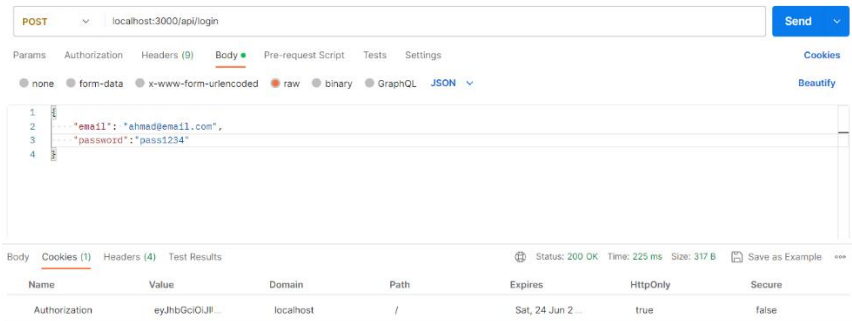
*Gambar 7 Status Running API*

Pada Gambar 43 merupakan keterangan apabila API sudah berjalan dengan menggunakan PORT 3000 dan siap untuk dipakai.



*Gambar 8 Register User*

Akses URL “localhost:3000/api/signup” untuk melakukan register seperti pada Gambar 44. Apabila pendaftaran berhasil maka respon akan dikembalikan dengan kode 200 Status OK.

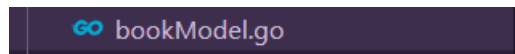


*Gambar 9 Login*

Setelah proses registrasi berhasil, selanjutnya adalah mencoba untuk Login menggunakan API dengan cara memasukan input dengan bentuk JSON pada Postman. Apabila User terdaftar maka API akan merespon dengan kode 200 dan token akan langsung disimpan dalam cookie.

## 5.2. Perancangan Endpoint Buku

### 1. Membuat Model Buku



*Gambar 10 File bookModel*

Hal yang pertama harus dilakukan pada saat akan merancang endpoint untuk buku adalah membuat file nya terlebih dahulu. Buat file dengan nama `bookModel.go` seperti pada Gambar 46.

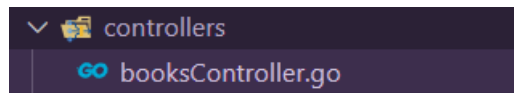
```
package models

type Books struct {
    ID          int `gorm:"primaryKey"`
    Judul       string
    Penulis     string
    Penerbit    string
    Tanggal_rilis string
}
```

*Kolom 17 Code Untuk Model Buku*

Selanjutnya adalah memasukkan kode seperti pada Kolom 19 pada file “bookModel.go”. Pada kode tersebut dilakukan terlebih dahulu pendeklarasian *package*, lalu dibuat struct untuk membuat struktur tabel sesuai dengan *field* dari buku yang akan diinputkan.

## 2. Membuat Controller Buku



*Gambar 11 Controller Buku*

Buatlah file seperti pada Gambar 47 yaitu file “booksController.go”, file ini digunakan untuk menampung berbagai *function* untuk *endpoint* yang akan dibuat nanti. Simpan file pada Controllers agar lebih mudah untuk manajemen filenya.



```

package controllers

import (
    "elib_v2/initializers"
    "elib_v2/models"

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

func Books(c *gin.Context) {
    var books []models.Books
    initializers.DB.Find(&books)

    c.JSON(200, books)
}

```

*Kolom 18 Code untuk Controller Buku*

Selanjutnya adalah memasukan kode pada file “booksController.go” agar dapat membuat komunikasi dengan *database*. Pertama adalah melakukan import terhadap *library* yang dibutuhkan. Lalu membuat *function* pertama untuk mengambil semua data buku seperti pada Kolom 20. Output dari buku akan diambil dengan tipe data *array* agar sejumlah tipe data dapat ditampung dalam satu tempat, apabila pengambilan data berhasil akan direspon dengan HTTP response kode 200 untuk sukses dan output buku akan dikeluarkan.

```

func Viewbookid(c *gin.Context) {
    id := c.Param("id")

    var books []models.Books
    initializers.DB.Find(&books, id)

    c.JSON(200, books)
}

```

*Kolom 19 Get Buku By ID*

Selanjutnya membuat *function* untuk mengambil buku berdasarkan ID seperti pada Kolom 21. Kolom ID akan ditampung dengan variabel *id* dan

parameter “id”. Variabel buku akan diambil dengan bentuk array dan hanya akan mengambil *field* “id” nya saja. Memanggil data buku hanya dengan id ini dibutuhkan untuk mengambil item secara spesifik untuk kebutuhan tertentu.

```
func AddBooks(c *gin.Context) {  
    // ambil data  
    var body struct {  
        Judul      string  
        Penulis    string  
        Penerbit    string  
        Tanggal_rilis string  
    }  
  
    c.Bind(&body)  
  
    //tambah buku  
    post := models.Books{Judul: body.Judul, Penulis: body.Penulis,  
        Penerbit: body.Penerbit, Tanggal_rilis: body.Tanggal_rilis}  
    result := initializers.DB.Create(&post)  
  
    if result.Error != nil {  
        c.Status(400)  
        return  
    }  
  
    c.JSON(200, gin.H{  
        "book": post,  
    })  
}
```

#### *Kolom 20 Function AddBooks*

Pada Kolom 22 merupakan function yang berfungsi untuk menambahkan data buku. *Field* buku akan ditampung terlebih dahulu ditampung dan di *bind* untuk mengubah data menjadi respon JSON, lalu field dari buku akan dideklarasikan ulang dalam variabel post dan variabel result akan memanggil initializer untuk memanggil method POST agar data bisa dimasukkan kedalam tabel, terakhir akan direspon dengan output JSON kode 200 apabila input buku berhasil.

```

func UpdateBook(c *gin.Context) {
    // ambil dulu id buku
    id := c.Param("id")

    // ambil data request
    var body struct {
        Judul      string
        Penulis     string
        Penerbit    string
        Tanggal_rilis string
    }

    c.Bind(&body)

    //ambil post update
    var books models.Books
    initializers.DB.First(&books, id)

    //update
    initializers.DB.Model(&books).Updates(models.Books{
        Judul:      body.Judul,
        Penulis:     body.Penulis,
        Penerbit:    body.Penerbit,
        Tanggal_rilis: body.Tanggal_rilis,
    })

    //respon
    c.JSON(200, gin.H{
        "book": books,
    })
}

```

#### Kolom 21 Function Update Buku

Pada Kolom 23 adalah *function* yang digunakan untuk melakukan perubahan terhadap data buku. Parameter buku akan dideklarasikan terlebih dahulu pada body, lalu buku akan diambil berdasarkan ID dan dipanggil oleh initialized menggunakan DB.first untuk memanggil *primary key* dari tabel buku tersebut, dan dideklarasikan kembali dalam function model agar dapat melakukan perubahan. Selanjutnya apabila perubahan berhasil akan direspon dengan HTTP response kode 200.

```
func DeleteBook(c *gin.Context) {
    id := c.Param("id")

    initializers.DB.Delete(&models.Books{}, id)

    //respon
    c.JSON(200, gin.H{
        "message": "Buku terhapus!",
    })
}
```

Kolom 22 Function Delete Book

Tahap selanjutnya adalah membuat function untuk menghapus buku seperti pada Kolom 24. Buku akan dipanggil berdasarkan input ID lalu dipanggil dalam function DB.Delete dan diisi oleh ID dari buku itu sendiri. Apabila proses menghapus buku berhasil maka akan direspon dengan HTTP Response dan message “buku terhapus”.

### 3. Pembuatan URL Endpoint

```
func main() {
    r := gin.Default()
    api := r.Group("/api")
    {
        //crud buku
        api.GET("/books", controllers.Books)
        api.GET("/books/:id", controllers.Viewbookid)
        api.POST("/books", controllers.AddBooks)
        api.PUT("/books/:id", controllers.UpdateBook)
        api.DELETE("/books/:id", controllers.DeleteBook)
    }
    r.Run()
}
```

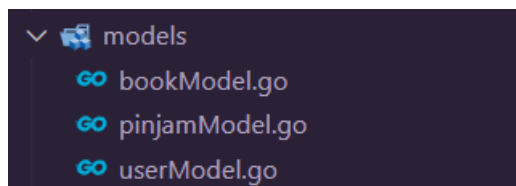
Kolom 23 Pembuatan Endpoint URL

Pada file “main.go” tambahkan kode untuk membuat rute URL *endpoint* agar dapat memanggil *function* yang telah dibuat. URL akan dibuat secara grup dalam “/api” dan sisanya URL spesifik hanya untuk buku. Terdapat beberapa HTTP *method* yang terdapat dalam URL yang pertama ada GET yang digunakan

untuk memanggil data, selanjutnya ada POST untuk menyimpan data, lalu ada PUT untuk mengubah data, dan terakhir ada DELETE untuk menghapus data. Masing-masing URL mempunyai rute tersendiri, URL yang mempunyai “:id” digunakan untuk memanggil data spesifik menggunakan *primary key* dari tabel buku.

### 5.3. Pembuatan Endpoint Pinjam Buku

#### 1. Membuat Model Pinjam



Gambar 12

Hal pertama yang harus dilakukan adalah membuat model untuk peminjaman buku seperti pada Gambar 48. Model dari pinjaman buku disimpan dalam folder models bersama dengan model-model lainnya.

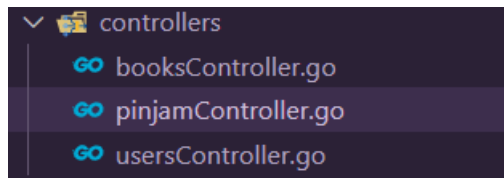
```
package models

type Pinjam struct {
    ID          int `gorm:"primaryKey"`
    UserID      int
    User        User
    BooksID     int
    Books       Books
    Tanggal_pinjam string
    Dikembalikan bool
}
```

Kolom 24 Model pinjam

Pada kolom 26 merupakan struktur dari model pinjam buku. Pada model tersebut terdapat beberapa *field* yang berhubungan dengan beberapa tabel yaitu UserID dan BooksID yang digunakan untuk mengambil *value* primary

key dari masing-masing tabel yaitu tabel User dan Books.



*Gambar 13 File Pinjam Controller*

Pada Gambar 49 merupakan nama dan lokasi dari file “pinjamController.go” yang disimpan bersamaan dengan model-model lainnya.

```
package controllers

import (
    "elib_v2/initializers"
    "elib_v2/models"

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

func Pinjam(c *gin.Context) {
    var pinjam []models.Pinjam
    initializers.DB.Find(&pinjam)

    c.JSON(200, pinjam)
}
```

*Kolom 25 Import dan Function Get All Pinjam*

Pada Kolom 27 adalah struktur awal dari controller untuk pinjaman buku. Pertama akan dilakukan *import* terlebih dahulu untuk library yang digunakan, lalu pembuatan *function* untuk pengambilan semua data pinjaman buku yang dideklarasikan dalam bentuk *array*. Lalu function tersebut akan memanggil initializer untuk berkomunikasi dengan database, apabila komunikasi berhasil dilakukan akan direspon dengan kode 200 yaitu StatusOK.

```
func GetPinjamid(c *gin.Context) {
    id := c.Param("id")

    var pinjam models.Pinjam
    initializers.DB.Find(&pinjam, id)

    c.JSON(200, pinjam)
}
```

Kolom 26 Get Pinjam By id

Selanjutnya pembuatan *function* untuk mengambil data pinjaman berdasarkan ID seperti pada Kolom 28. Pada *function* ini data akan dipanggil menggunakan initializer dan diambil field id nya saja.

```
func AddPinjam(c *gin.Context) {
    // ambil data
    var body struct {
        UserID      int
        BooksID      int
        Tanggal_pinjam string
        Dikembalikan bool
    }

    c.Bind(&body)

    //tambah buku
    pinjam := models.Pinjam{UserID: body.UserID, BooksID: body.BooksID,
    Tanggal_pinjam: body.Tanggal_pinjam, Dikembalikan: body.Dikembalikan}
    result := initializers.DB.Create(&pinjam)

    if result.Error != nil {
        c.Status(400)
        return
    }

    c.JSON(200, gin.H{
        "pinjaman": pinjam,
    })
}
```

Kolom 27 AddPinjam

Pada Kolom 29 merupakan kolom untuk menambahkan data pinjaman buku pada tabel pinjam. *Field* terisi dengan beberapa value *foreign key* yang diambil dari *primary key* yang terdapat pada tabel User dan Books. Lalu

dideklarasikan pada variabel pinjam dan di-assign dalam array. apabila terdapat error pada saat melakukan input, maka akan menampilkan kode 400 Status Bad Request, sedangkan apabila input berhasil akan menampilkan kode 200 StatusOK.

```
func UpdatePinjam(c *gin.Context) {
    id := c.Param("id")

    // ambil data request
    var body struct {
        UserID      int
        BooksID     int
        Tanggal_pinjam string
        Dikembalikan bool
    }

    c.Bind(&body)

    //ambil post update
    var pinjam models.Pinjam
    initializers.DB.First(&pinjam, id)

    //update
    initializers.DB.Model(&pinjam).Updates(models.Pinjam{
        UserID:      body.UserID,
        BooksID:     body.BooksID,
        Tanggal_pinjam: body.Tanggal_pinjam,
        Dikembalikan: body.Dikembalikan,
    })

    //respon
    c.JSON(200, gin.H{
        "message": pinjam,
    })
}
```

Kolom 28 UpdatePinjam

Pada Kolom 30 merupakan *function* yang digunakan untuk mengubah data peminjaman buku, *field* yang digunakan untuk mengubah data peminjaman buku sama seperti yang digunakan pada penambahan buku yaitu beberapa *foreign key* dll. *Field* id pada tabel pinjam akan dipanggil untuk mengubah data spesifik pada kolom tertentu.



```
func DeletePinjam(c *gin.Context) {  
    id := c.Param("id")  
  
    initializers.DB.Delete(&models.Pinjam{}, id)  
  
    //respon  
    c.JSON(200, gin.H{  
        "message": "Pinjaman terhapus!",  
    })  
}
```

*Kolom 29 Function DeletePinjam*

Pada Kolom 31 merupakan *function* yang digunakan untuk menghapus data pinjaman buku. ID pinjaman akan dipanggil terlebih dahulu untuk memasukan data tertentu kemudian apabila proses penghapusan berhasil maka akan direspon dengan kode 200 dengan statusOK dan message dalam format JSON yaitu “pinjaman terhapus!”.