

Go Programming Secure Your Go Apps

Chapter 3 Sesi 2 : MIDDLEWARE





Middleware

Middleware

Middleware merupakan sebuah fungsi yang akan ter - eksekusi sesudah maupun sebelum mencapai sebuah endpoint. Biasa middleware digunakan untuk logging atau untuk mengamankan sebuah endpoint seperti contohnya proses autentikasi dan otorisasi.

Untuk membuat middleware pada bahasa Go, kita akan menggunakan package *net/http* dengan menggunakan multiplexer nya agar kita dapat melakukan kustomisasi.

Gambar dibawah ini merupakan gambaran dari alur sebuah middleware.

```
Router => Middleware Handler => Application Handler
```

Middleware

Untuk membuat sebuah middleware, maka kita harus suatu function yang memuaskan interface *http.Handler* yang dimana interface ini memiliki satu buah method dengan berupa **`ServeHTTP(http.ResponseWriter, *http.Request)`**.

Perhatikan pada gambar diatas. Function *middleware1* merupakan sebuah middleware.

Setiap middleware pada bahasa Go akan menerima satu parameter dengan tipe data *http.Handler* dan harus mereturn tipe data *http.Handler*.

Jika kita perhatikan pada gambar diatas, function *middleware1* mereturn sebuah anonymous function dengan nama *http.HandlerFunc* yang dimana function ini menerima satu argument berupa tipe data function yang mempunyai argument yang sama seperti method *ServeHTTP*.

```
func middleware1(next http.Handler) http.Handler {  
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {  
        next.ServeHTTP(w, r)  
    })  
}
```

Middleware

Perlu diingat bahwa function *http.HandlerFunc* secara default mengimplementasikan method *ServeHTTP* dan setiap function yang mempunyai skema parameter yang sama dengan method *ServeHTTP* dapat di ubah tipe datanya menjadi interface *http.Handler* dengan cara menjadikannya parameter dari function *http.HandlerFunc*.

```
func middleware1(next http.Handler) http.Handler {  
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {  
        next.ServeHTTP(w, r)  
    })  
}
```

Middleware

Perhatikan pada gambar disebelah kanan. Pada line 11, kita mendeklarasikan sebuah variable bernama *mux* yang akan menjadi *multiplexer* nya.

Kemudian pada line 13, kita membuat variable bernama *endpoint* yang menampung satu-satunya endpoint pada aplikasi kita saat ini yaitu function *greet*. Perhatikan bahwa kita menjadikan function *greet* sebagai argument dari function *http.HandlerFunc* agar function *greet* dapat menjadi sebuah function dengan tipe data *http.Handler*.

Kemudian pada line 15, kita membuat routingsnya dengan menggunakan method *Handle* dari pointer struct *ServeMux*. Method *Handle* menerima 2 parameter berupa string dari route path nya, dan handler nya yang bertipe *http.Handler*. Lalu perhatikan bahwa kita menggunakan 2 middleware sebelum kita dapat mencapai function *greet* yang merupakan endpointnya.

```
8
9 func main() {
10
11     mux := http.NewServeMux()
12
13     endpoint := http.HandlerFunc(greet)
14
15     mux.Handle("/", middleware1(middleware2(endpoint)))
16
17     fmt.Println("Listening to port 8080")
18
19     err := http.ListenAndServe(":3000", mux)
20
21     log.Fatal(err)
22 }
23
24 func greet(w http.ResponseWriter, r *http.Request) {
25     w.Write([]byte("Hello World!!!"))
26 }
27
28 func middleware1(next http.Handler) http.Handler {
29
30     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
31         fmt.Println("middleware pertama")
32         next.ServeHTTP(w, r)
33     })
34 }
35
36 func middleware2(next http.Handler) http.Handler {
37     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
38         fmt.Println("middleware kedua")
39         next.ServeHTTP(w, r)
40     })
41 }
42
```

Middleware

Sekarang mari kita jalankan server dari aplikasi kita, dan buka browser dan arahkan kepada url <http://localhost:3000/>.

Jika sudah, maka kita akan melihat logging yang sudah kita buat pada ke-2 middleware kita seperti pada gambar pertama disebelah kanan.

Dan pada browser kita akan terlihat seperti pada gambar kedua.

```
Listening to port 8080
middleware pertama
middleware kedua
```

← → ↻ ⓘ localhost:3000

Hello World!!!



Securing Our App With JSONWebToken

Securing Our App With JSONWEBTOKEN

Kita akan belajar mengamankan Rest API kita dengan menggunakan *JsonWebToken*. *JsonWebToken* adalah token berbentuk string yang digunakan untuk pertukaran informasi antara 2 belah pihak atau client dan server.

JsonWebToken biasa digunakan pada aplikasi web yang berbasis SPA atau Single Page Application dengan alur sebagai berikut:

- User melakukan login melalui client, biasanya user hanya perlu menginput email dan password saja.
- Setelah user memencet tombol submit, maka client akan mengirimkan data yang di input oleh user kepada server.
- Setelah data diterima oleh server, maka server akan memeriksa terlebih dahulu apakah data yang dikirimkan oleh client merupakan data yang valid atau tidak.
- Jika datanya valid, maka server akan mengirimkan data user tersebut kepada client, namun data user nya telah dijadikan sebuah token dengan menggunakan *JWT* (*JsonWebToken*).
- Lalu client akan menyimpan token yang dikirimkan oleh server pada local storage atau session storage, dan jika client hendak mengirimkan request kepada server kembali, maka client perlu mengirimkan token tersebut kepada server melalui request headers.
- Lalu ketika token tersebut sudah oleh server, maka server akan melakukan proses autentikasi terhadap token tersebut yang dikirimkan oleh client.

Berikut merupakan link resmi dari JWT <https://jwt.io/>

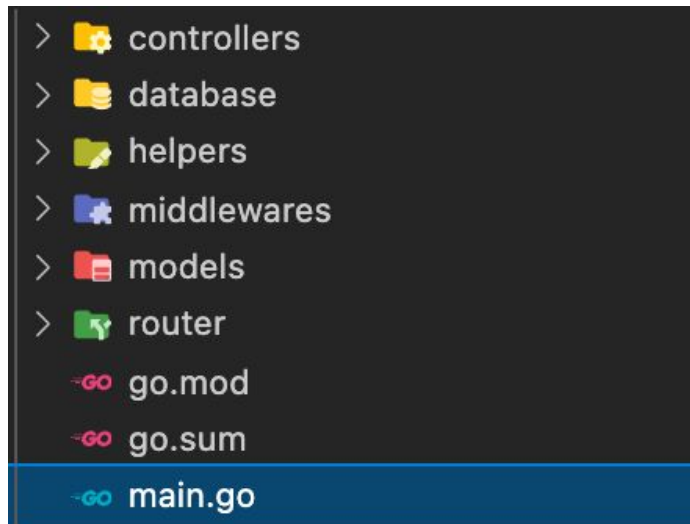
Setting Up Workspace

Pada kali ini, kita akan membuat suatu Rest API yang sangat simple, yang dimana kita akan membuat 2 table saja yaitu table *User* dan *Product*. Agar lebih memudahkan kita pada saat membuat aplikasinya, maka kita akan membutuhkan bantuan dari ORM **Gorm** yang akan memudahkan kita dalam berkomunikasi dengan database dan framework **Gin** sebagai routingnya.

Maka dari buatlah sebuah folder dengan nama **go-jwt**. Lalu jalankan perintah **go mod init go-jwt** pada terminal dan pastikan terminal kita di dalam folder **go-jwt**. Lalu setelah itu buatlah sebuah file dengan nama **main.go**.

Lalu di dalam folder **go-jwt**, buatlah folder-folder dengan nama **controllers**, **models**, **helpers**, **router**, **database** dan **middlewares**.

Jika sudah terbuat semuanya, maka sekarang folder tree kita akan terlihat seperti pada gambar di sebelah kanan.



Setting Up Workspace

Kemudian jalankanlah perintah-perintah dibawah pada terminal untuk menginstall package-package yang kita perlukan.

```
go get github.com/asaskevich/govalidator
go get github.com/dgrijalva/jwt-go
go get github.com/gin-gonic/gin
go get golang.org/x/crypto
go get gorm.io/driver/postgres
go get gorm.io/gorm
```

Setting Up Database Configuration

Didalam folder **models**, buatlah 3 file dengan nama **gormModel.go**, **user.go**, dan **product.go**.

Jika sudah, selanjutnya ikutilah syntax pada gambar dibawah ini pada file **gormModel.go**.

```
package models

import "time"

type GormModel struct {
    ID          uint          `gorm:"primaryKey" json:"id"`
    CreatedAt   *time.Time    `json:"created_at,omitempty"`
    UpdatedAt   *time.Time    `json:"updated_at,omitempty"`
}
```

Setting Up Database Configuration

Kemudian ikutilah syntax dibawah ini untuk file **user.go**.

```
1 package models
2
3 import (
4     "github.com/asaskevich/govalidator"
5     "gorm.io/gorm"
6 )
7
8 type User struct {
9     GormModel
10    FullName string `gorm:"not null" json:"full_name" form:"full_name" valid:"required~Your full name is required"`
11    Email     string  `gorm:"not null;uniqueIndex" json:"email" form:"email" valid:"required~Your email is required,email~Invalid email format"`
12    Password string  `gorm:"not null" json:"password" form:"password" valid:"required~Your password is required,minstringlength(6)~Password has to have a minimum length of 6 characters"`
13    Products []Product `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;" json:"products"`
14 }
15
16 func (u *User) BeforeCreate(tx *gorm.DB) (err error) {
17     _, errCreate := govalidator.ValidateStruct(u)
18
19     if errCreate != nil {
20         err = errCreate
21         return
22     }
23
24     err = nil
25     return
26 }
27 }
```

Setting Up Database Configuration

```
1 package models
2
3 import (
4     "github.com/asaskevich/govalidator"
5     "gorm.io/gorm"
6 )
7
8 type User struct {
9     GormModel
10    FullName string `gorm:"not null" json:"full_name" form:"full_name" valid:"required~Your full name is required"`
11    Email     string  `gorm:"not null;uniqueIndex" json:"email" form:"email" valid:"required~Your email is required,email~Invalid email format"`
12    Password string  `gorm:"not null" json:"password" form:"password" valid:"required~Your password is required,minstringlength(6)~Password has to have a minimum length of 6 characters"`
13    Products []Product `gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET NULL;" json:"products"`
14 }
```

Jika kita lihat pada gambar diatas, terdapat *tag-tag* dengan penulisan **valid**. Penulisan **valid** merupakan cara kita dalam menentukan validasi terhadap field-field untuk table *User* menggunakan package **govalidator**. Lalu validasinya kita akfitkan didalam hooks *beforeCreate* dari orm **Gorm**. Jika teman-teman ingin mengeksplorasi lebih jauh tentang pacakge **govalidator**, maka bisa mengunjungi langsung kepada dokumentasi aslinya pada URL <https://github.com/asaskevich/govalidator>.

Setting Up Database Configuration

Kemudian ikutilah syntax berikut untuk file **product.go**.

```
1 package models
2
3 import (
4     "github.com/asaskevich/govalidator"
5     "gorm.io/gorm"
6 )
7
8 type Product struct {
9     GormModel
10    Title      string `json:"title" form:"title" valid:"required-Title of your product is required"`
11    Description string `json:"description" form:"description" valid:"required-Description of your product is required"`
12    UserID      uint
13    User        *User
14 }
15
16 func (p *Product) BeforeCreate(tx *gorm.DB) (err error) {
17     _, errCreate := govalidator.ValidateStruct(p)
18
19     if errCreate != nil {
20         err = errCreate
21         return
22     }
23
24     err = nil
25     return
26 }
27
28 func (p *Product) BeforeUpdate(tx *gorm.DB) (err error) {
29     _, errCreate := govalidator.ValidateStruct(p)
30
31     if errCreate != nil {
32         err = errCreate
33         return
34     }
35
36     err = nil
37     return
38 }
39 }
```

Setting Up Database Configuration

Kemudian pada folder **database**, buat lah satu file didalamnya dengan nama **db.go**.

```
1 package database
2
3 import (
4     "fmt"
5     "go-jwt/models"
6     "log"
7
8     "gorm.io/driver/postgres"
9     "gorm.io/gorm"
10 )
11
12 var (
13     host    = "localhost"
14     user    = "postgres"
15     password = "postgres"
16     dbPort  = "5432"
17     dbname  = "simple-api"
18     db      *gorm.DB
19     err     error
20 )
21
22 func StartDB() {
23     config := fmt.Sprintf("host=%s user=%s password=%s dbname=%s port=%s sslmode=disable", host, user, password, dbname, dbPort)
24     dsn := config
25     db, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
26
27     if err != nil {
28         log.Fatal("error connecting to database :", err)
29     }
30
31     fmt.Println("sukses koneksi ke database")
32     db.Debug().AutoMigrate(models.User{}, models.Product{})
33
34 }
35
36 func GetDB() *gorm.DB {
37     return db
38 }
```


Setting Up Database Configuration

Kemudian pada folder **database**, buat lah satu file didalamnya dengan nama **db.go**.

Perlu diperhatikan disini bahwa nilai yang diberikan pada variable *password* dan *user* dapat berbeda-beda tergantung dari pada saat konfigurasi yang kita berikan pertama kali pada saat menginstall *Postgresql*.

```
1 package database
2
3 import (
4     "fmt"
5     "go-jwt/models"
6     "log"
7
8     "gorm.io/driver/postgres"
9     "gorm.io/gorm"
10 )
11
12 var (
13     host    = "localhost"
14     user    = "postgres"
15     password = "postgres"
16     dbPort  = "5432"
17     dbname  = "simple-api"
18     db      *gorm.DB
19     err      error
20 )
21
22 func StartDB() {
23     config := fmt.Sprintf("host=%s user=%s password=%s dbname=%s port=%s sslmode=disable", host, user, password, dbname, dbPort)
24     dsn := config
25     db, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
26
27     if err != nil {
28         log.Fatal("error connecting to database :", err)
29     }
30
31     fmt.Println("sukses koneksi ke database")
32     db.Debug().AutoMigrate(models.User{}, models.Product{})
33
34 }
35
36 func GetDB() *gorm.DB {
37     return db
38 }
```

Create Bcrypt For Password Hasing

Pada folder **helpers**, buatlah satu file dengan nama **bcrypt.go**. Kemudian isilah dengan syntax dibawah ini pada file tersebut. Function *HashPass* akan kita gunakan untuk meng-hashing password user sebelum disimpan kedalam database, dan function *ComparePass* digunakan untuk mengkomparasi password user yang sudah dihash dengan password user yang di input ketika sedang melakukan login.

```
1  package helpers
2
3  import "golang.org/x/crypto/bcrypt"
4
5  func HashPass(p string) string {
6      salt := 8
7      password := []byte(p)
8      hash, _ := bcrypt.GenerateFromPassword(password, salt)
9
10     return string(hash)
11 }
12
13 func ComparePass(h, p []byte) bool {
14     hash, pass := []byte(h), []byte(p)
15
16     err := bcrypt.CompareHashAndPassword(hash, pass)
17
18     return err == nil
19 }
```

Create Bcrypt For Password Hasing

Setelah itu, agar proses hashing passwordnya lebih mudah, maka kita dapat menggunakannya pada hooks *beforeCreate* pada file **user.go** di dalam folder **models** seperti pada line 26 pada gambar di sebelah kanan.

```
1 package models
2
3 import (
4     "go-jwt/helpers"
5
6     "github.com/asaskevich/govalidator"
7     "gorm.io/gorm"
8 )
9
10 type User struct {
11     GormModel
12     FullName string `gorm:"not null" json:"full_name" `
13     Email string `gorm:"not null;uniqueIndex" json:"email format"`
14     Password string `gorm:"not null" json:"password" `
15     Products []Product `gorm:"constraint:OnUpdate:CASCADE" `
16 }
17
18 func (u *User) BeforeCreate(tx *gorm.DB) (err error) {
19     _, errCreate := govalidator.ValidateStruct(u)
20
21     if errCreate != nil {
22         err = errCreate
23         return
24     }
25
26     u.Password = helpers.HashPass(u.Password)
27     err = nil
28     return
29 }
30
31 }
```

Create Helper To Content-Type

Agar nantinya client dapat mengirimkan request body melalui data *JSON* ataupun melalui *form*, maka sekarang kita akan membuat sebuah function yang hanya akan digunakan untuk mendapat Content Type dari request headers yang dikirimkan oleh client.

Maka dari itu, buatlah satu file didalam folder **helpers** dengan nama **headerValue.go**. Kemudian ikutilah syntax dibawah ini pada file tersebut.

```
1 package helpers
2
3 import (
4     | "github.com/gin-gonic/gin"
5 )
6
7 func GetContentType(c *gin.Context) string {
8     | return c.Request.Header.Get("Content-Type")
9 }
```

User Registration Endpoint

Pada folder **controllers**, buatlah satu file dengan nama **userControllers.go**. Kemudian buatlah satu function yang digunakan sebagai endpoint untuk registrasi user.

Maka dari itu ikutilah syntax di sebelah kanan pada file **userControllers.go**.

Jika kita melihat pada line 23 dan line 25, kita menggunakan method *ShouldBindJSON* dan *ShouldBind*.

Kedua method tersebut merupakan method-method dari framework **Gin** yang digunakan untuk mendapatkan data dari request body yang dikirimkan oleh client, lalu setelah itu data-data tersebut disimpan kedalam sebuah struct.

ShouldBindJson digunakan untuk mendapatkan data request yang berbentuk JSON , sedangkan *ShouldBind* digunakan untuk mendapatkan data yang dikirimkan melalui *form*.

```
1 package controllers
2
3 import (
4     "go-jwt/database"
5     "go-jwt/helpers"
6     "go-jwt/models"
7     "net/http"
8
9     "github.com/gin-gonic/gin"
10 )
11
12 var (
13     appJSON = "application/json"
14 )
15
16 func UserRegister(c *gin.Context) {
17     db := database.GetDB()
18     contentType := helpers.GetContentType(c)
19     _, _ = db, contentType
20     User := models.User{}
21
22     if contentType == appJSON {
23         c.ShouldBindJSON(&User)
24     } else {
25         c.ShouldBind(&User)
26     }
27
28     err := db.Debug().Create(&User).Error
29
30     if err != nil {
31         c.JSON(http.StatusBadRequest, gin.H{
32             "error":    "Bad Request",
33             "message":  err.Error(),
34         })
35         return
36     }
37
38     c.JSON(http.StatusCreated, gin.H{
39         "id":      User.ID,
40         "email":   User.Email,
41         "full_name": User.FullName,
42     })
43 }
```

User Registration Endpoint

Pada folder **router**, buatlah satu file bernama **router.go**. Kemudian ikutilah syntax seperti pada gambar dibawah ini pada file tersebut.

```
1 package router
2
3 import (
4     "go-jwt/controllers"
5
6     "github.com/gin-gonic/gin"
7 )
8
9 func StartApp() *gin.Engine {
10     r := gin.Default()
11
12     userRouter := r.Group("/users")
13     {
14         userRouter.POST("/register", controllers.UserRegister)
15     }
16
17     return r
18 }
19 }
```

User Registration Endpoint

Method *Group* pada line 12 merupakan sebuah method dari framework *Gin* yang akan sangat mempermudah kita dalam melakukan route grouping. Maka nantinya ketika client mengirimkan request dengan path **/users**, maka request tersebut akan masuk kedalam scope pada line 13 hingga 14. Scope tersebut akan dikhususkan untuk routing endpoint users.

```
1 package router
2
3 import (
4     "go-jwt/controllers"
5
6     "github.com/gin-gonic/gin"
7 )
8
9 func StartApp() *gin.Engine {
10     r := gin.Default()
11
12     userRouter := r.Group("/users")
13     {
14         userRouter.POST("/register", controllers.UserRegister)
15     }
16
17     return r
18 }
19 }
```

Starting The Application

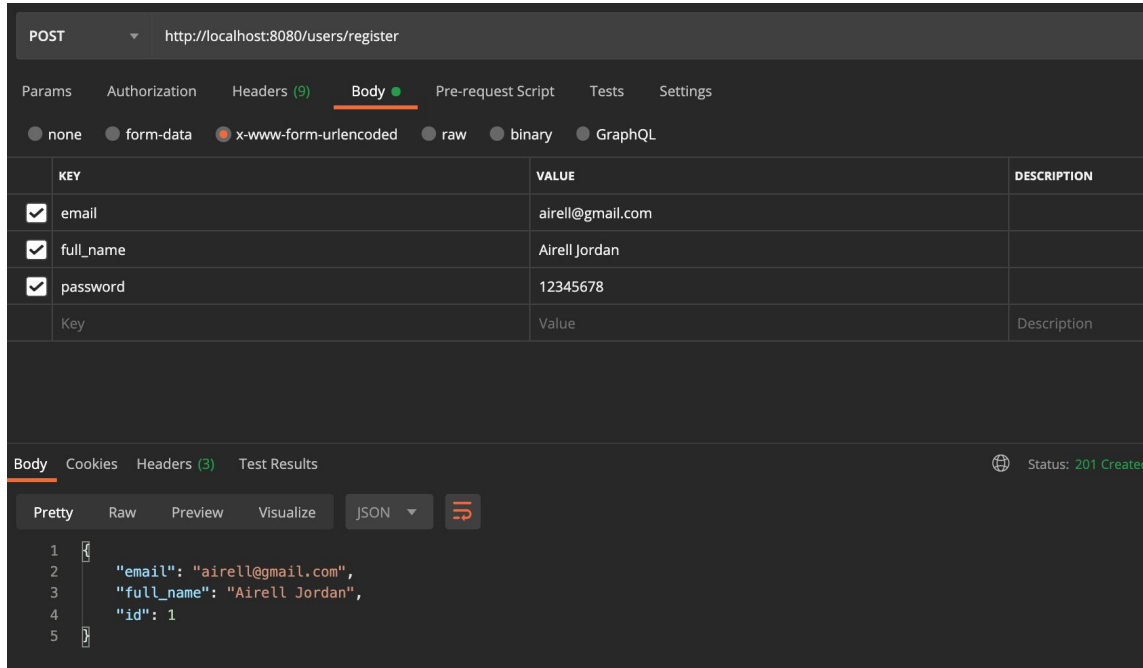
Ikutilah syntax dibawah ini pada file **main.go**.

```
1  package main
2
3  import (
4      "go-jwt/database"
5      "go-jwt/router"
6  )
7
8  func main() {
9      database.StartDB()
10     r := router.StartApp()
11     r.Run(":8080")
12 }
```

Lalu jalankan aplikasi kita melalui terminal dengan perintah **go run main.go**. Maka aplikasi kita sudah akan berjalan, dan jika melihat pada database kita, maka seluruh table yang kita perlukan beserta field-fieldnya telah otomatis termigrasi oleh orm **Gorm**.

Test User Registration Endpoint With Postman

Sekarang kita akan mencoba untuk mendaftarkan satu data user melalui *Postman*. Maka dari itu buatlah sebuah request seperti pada gambar dibawah ini dengan method POST.



Sekarang jika kita periksa pada database kita, kita sudah mempunyai satu buah data user dengan email `airell@gmail.com`.

Create Helper For Generating Token

Sekarang kita akan membuat sebuah function yang akan kita gunakan untuk membuat token JWT. Maka dari itu pada folder **helpers**, buatlah satu file dengan nama **jwt.go**, dan ikutilah syntax dibawah untuk file tersebut.

```
1 package helpers
2
3 import (
4     "github.com/dgrijalva/jwt-go"
5 )
6
7 var secretKey = "rahasia"
8
9 func GenerateToken(id uint, email string) string {
10     claims := jwt.MapClaims{
11         "id": id,
12         "email": email,
13     }
14
15     parseToken := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
16
17     signedToken, _ := parseToken.SignedString([]byte(secretKey))
18
19     return signedToken
20 }
```

Create Helper For Generating Token

Mari kita bedah arti dari syntax dihalaman sebelumnya.

- Karena token *JWT* yang akan dikirimkan oleh server kepada client akan berisikan data user, maka kita perlu menyimpannya dengan menggunakan *jwt.MapClaims* yang merupakan sebuah tipe data *map[string]interface{}* yang berasal dari package **jwt-go**. Perlu diingat disini bahwa kita tidak diperkenankan untuk meyimpan data-data user yang bersifat kredensial seperti password, pin atm dan lain-lain. Untuk kali ini kita hanya perlu menyimpan *id* dan *email* dari user.
- Line 15, menentukan metode enkripsi yang ingin kita gunakan. Pada kali ini kita akan menggunakan metode enkripsi **HS256**. Method *jwt.NewWithClaims* akan digunakan untuk memasukkan data-data user yang kita simpan pada *jwt.MapClaims* dan sekaligus menentukan metode enkripsinya. Nantinya method ini akan mengembalikan sebuah nilai dengan struct pointer dari *jwt.Token*.
- Pada line 17, kita menggunakan method *jwt.SignedString* yang merupakan sebuah method dari struct pointer *jwt.Token*. Method ini digunakan untuk parsing token menjadi sebuah string panjang yang nantinya akan dikirimkan oleh server kepada client. Method ini menerima satu parameter berupa sebuah **secret key**. **Secret key** merupakan data yang sangat kredensial karena akan digunakan untuk mengautentikasi token yang nanti nya akan dikirimkan oleh client kepada server ketika client ingin mengirimkan sebuah request yang membutuhkan proses autentikasi token. **Secret key** ketika membuat token harus sama dengan **secret key** yang digunakan untuk mengautentikasi tokennya, jika tidak sama maka token akan dianggap tidak valid dan akan menghasilkan error.

User Login Endpoint

Pada file **UserController.go** yang terdapat didalam folder **controllers**, buatlah sebuah function untuk keperluan endpoint user login seperti pada gambar di halaman berikutnya.

Nantinya ketika user hendak melakukan login, maka user hanya perlu menginput email dan passwordnya saja melalui client.

Kemudian ketika email dan password sudah dikirimkan oleh client kepada server dan sudah diteima oleh server, maka server akan mencoba untuk mendapat satu data dari database berdasarkan email yang dikirimkan oleh client.

Jika memang ada, maka user akan mencoba untuk melakukan komparasi antara password user yang telah di hashing dan password user yang input melalui client.

Jika proses komparasi nya berhasil, maka server akan langsung membuat token dan akan dikirimkan kepada client.

User Login Endpoint

```
45 func UserLogin(c *gin.Context) {
46     db := database.GetDB()
47     contentType := helpers.GetContentType(c)
48     _, _ = db, contentType
49     User := models.User{}
50     password := ""
51
52     if contentType == appJSON {
53         c.ShouldBindJSON(&User)
54     } else {
55         c.ShouldBind(&User)
56     }
57
58     password = User.Password
59
60     err := db.Debug().Where("email = ?", User.Email).Take(&User).Error
61
62     if err != nil {
63         c.JSON(http.StatusUnauthorized, gin.H{
64             "error": "Unauthorized",
65             "message": "invalid email/password",
66         })
67         return
68     }
69 }
```

```
70 comparePass := helpers.ComparePass([]byte(User.Password), []byte(password))
71
72 if !comparePass {
73     c.JSON(http.StatusUnauthorized, gin.H{
74         "error": "Unauthorized",
75         "message": "invalid email/password",
76     })
77     return
78 }
79
80 token := helpers.GenerateToken(User.ID, User.Email)
81
82 c.JSON(http.StatusOK, gin.H{
83     "token": token,
84 })
85
86 }
```

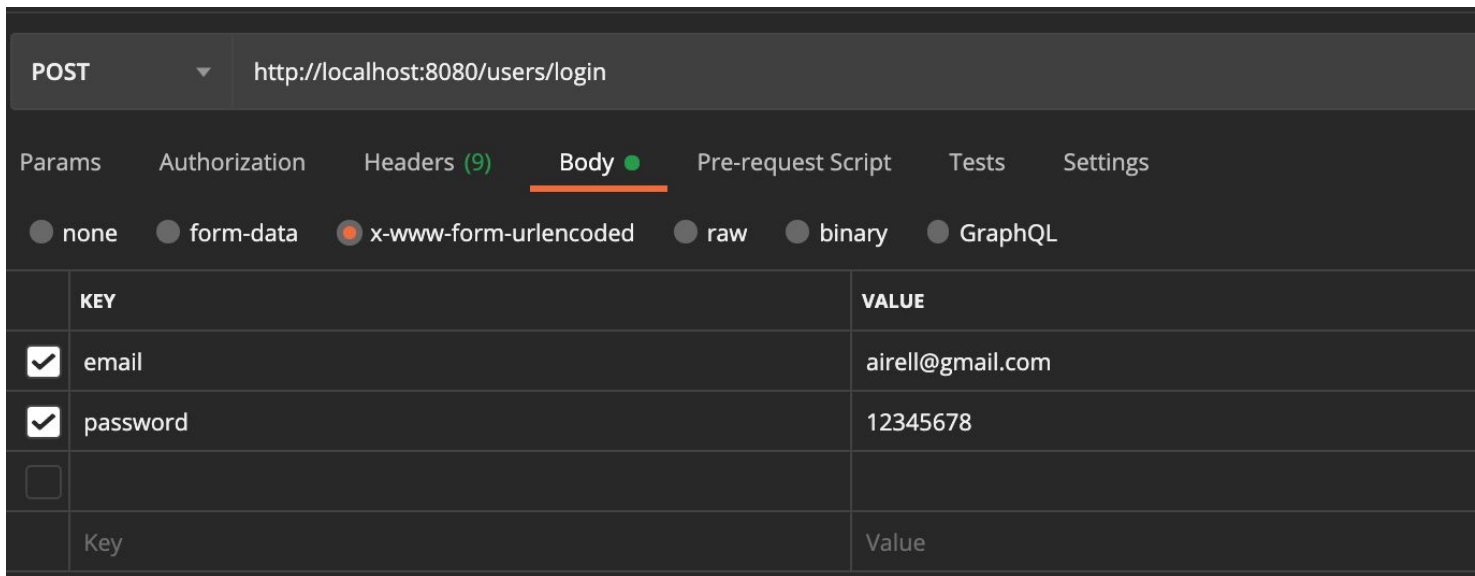
User Login Router

Sekarang mari kita tambahkan satu buat route yang akan mengarah pada controller *UserLogin*. Untuk itu mari kita tambahkan sebuah route pada file **router.go** yang terdapat di dalam folder **router**. Caranya seperti pada gambar dibawah ini.

```
func StartApp() *gin.Engine {  
    r := gin.Default()  
  
    userRouter := r.Group("/users")  
    {  
        userRouter.POST("/register", controllers.UserRegister)  
        userRouter.POST("/login", controllers.UserLogin)  
    }  
  
    return r  
}
```

Test User Login Endpoint With Postman

Mari kita buat sebuah request untuk mencoba endpoint user login yang baru saja kita buat. Buatlah sebuah request seperti pada gambar dibawah ini.



POST http://localhost:8080/users/login

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	KEY	VALUE
<input checked="" type="checkbox"/>	email	airell@gmail.com
<input checked="" type="checkbox"/>	password	12345678
<input type="checkbox"/>		
	Key	Value

Test User Login Endpoint With Postman

Setelah request telah berhasil terkirim, maka kita akan mendapatkan response dari server berupa token seperti pada gambar dibawah ini.

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFpcmVsbnBjb20iLCJpZCI6MX0.4Dv5YHe92K_duuFhn8ZmE-UwDkcLt-phFKRBDgBrYo"  
3 }
```


Create Helper To Validate Token

Sekarang kita akan membuat function bernama *VerifyToken* yang akan kita buat pada file *jwt.go* di dalam folder *helpers*. Function *VerifyToken* tersebut akan kita gunakan melakukan validasi terhadap token yang akan dikirimkan kembali oleh client. Caranya seperti pada gambar dibawah ini.

```
26 func VerifyToken(c *gin.Context) (interface{}, error) {
27     errReponse := errors.New("sign in to proceed")
28     headerToken := c.Request.Header.Get("Authorization")
29     bearer := strings.HasPrefix(headerToken, "Bearer")
30
31     if !bearer {
32         return nil, errReponse
33     }
34
35     stringToken := strings.Split(headerToken, " ")[1]
36
37     token, _ := jwt.Parse(stringToken, func(t *jwt.Token) (interface{}, error) {
38         if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {
39             return nil, errReponse
40         }
41         return []byte(secretKey), nil
42     })
43
44     if _, ok := token.Claims.(jwt.MapClaims); !ok && !token.Valid {
45         return nil, errReponse
46     }
47
48     return token.Claims.(jwt.MapClaims), nil
49 }
50
```

Create Helper To Validate Token

Mari kita bedah maksud dari syntax pada halaman sebelumnya:

- Pada line 27, kita membuat sebuah variable bernama *errResponse* yang dimana kita berikan nilai dengan sebuah custom error message menggunakan method *New* dari package *errors*.
- Pada line 28, kita mencoba untuk mendapatkan nilai dari variable *Authorization* yang terletak pada request headers yang dikirimkan oleh client. Karena pada nantinya, ketika client ingin mengirimkan sebuah request kepada endpoint di dalam server yang dimana endpoint tersebut memerlukan proses autentikasi token, maka client harus mengirimkan token kepada server dan menyimpannya di dalam headers pada variable bernama **Authorization**.
- Pada line 29 - 33, kita mencoba memeriksa jika token yang dikirimkan oleh client memiliki prefix **Bearer**. Jika tidak maka kita akan mengirimkan error karena akan dianggap sebagai token yang tidak valid. Karena token yang harus dikirimkan oleh client kepada server adalah sebuah **Bearer token** atau sebuah token yang mempunyai prefix **Bearer**.
- Pada line 35, kita mencoba untuk mengambil token nya tanpa prefix **Bearer** nya.

Create Helper To Validate Token

- Pada line 37 - 42, kita mencoba untuk memarsing tokennya menjadi sebuah struct pointer dari *jwt.Token*. Kemudian kita juga memeriksa apakah metode enkripsi dari tokennya adalah metode **HS256** dengan cara mengcasting metodenya menjadi tipe data pointer dari struct *jwt.SigningMethodHMAC*.
- Pada line 44, kita memeriksa apakah ketika kita mengcasting claim dari tokennya menjadi tipe data *jwt.MapClaims* akan menghasilkan error atau tidak. Kita juga sekaligus memeriksa apakah tokennya merupakan token yang valid atau tidak.
- Pada line 48, kita me-return claim dari tokennya yang dimana claim tersebut berisikan data yang kita simpan pada tokennya ketika pertama kali dibuat. Isinya adalah *email* dan *id* user yang sudah berhasil melakukan login.

Create Authentication Middleware

Sekarang kita akan membuat middleware untuk proses autentikasinya. Buat satu file dengan nama **authentication.go** didalam folder **middlewares**. Kemudian ikutilah syntax pada gambar dibawah ini untuk file tersebut.

```
10 func Authentication() gin.HandlerFunc {
11     return func(c *gin.Context) {
12         verifyToken, err := helpers.VerifyToken(c)
13         _ = verifyToken
14
15         if err != nil {
16             c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{
17                 "error": "Unauthenticated",
18                 "message": err.Error(),
19             })
20             return
21         }
22         c.Set("userData", verifyToken)
23         c.Next()
24     }
25 }
26
```

Create Authentication Middleware

Bisa dilihat pada gambar di sebelah kanan, pada line 12 kita memanggil function *VerifyToken* yang telah kita buat sebelumnya pada package **helpers**.

Lalu pada line 22, kita memanggil method *Set* yang berasal dari framework **Gin** untuk menyimpan claim dari tokennya kedalam data request agar dapat diambil pada endpoint berikutnya.

Lalu pada line 23 kita menggunakan method *Next* agar dapat melanjutkan proses keapda endpoint berikutnya.

```
10 func Authentication() gin.HandlerFunc {
11     return func(c *gin.Context) {
12         verifyToken, err := helpers.VerifyToken(c)
13         _ = verifyToken
14
15         if err != nil {
16             c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{
17                 "error": "Unauthenticated",
18                 "message": err.Error(),
19             })
20             return
21         }
22         c.Set("userData", verifyToken)
23         c.Next()
24     }
25 }
26
```

Create Product Endpoint

Sekarang kita akan membuat suatu endpoint untuk membuat data product baru.

Untuk itu pada folder **controllers**, buat lah satu file dengan nama **product.go** dan buatlah satu function dengan nama *CreateProduct* pada file tersebut.

Caranya seperti pada gambar di sebelah kanan.

```
13 func CreateProduct(c *gin.Context) {
14     db := database.GetDB()
15     userData := c.MustGet("userData").(jwt.MapClaims)
16     contentType := helpers.GetContentType(c)
17
18     Product := models.Product{}
19     userID := uint(userData["id"].(float64))
20
21     if contentType == appJSON {
22         c.ShouldBindJSON(&Product)
23     } else {
24         c.ShouldBind(&Product)
25     }
26
27     Product.UserID = userID
28
29     err := db.Debug().Create(&Product).Error
30
31     if err != nil {
32         c.JSON(http.StatusBadRequest, gin.H{
33             "err": "Bad Request",
34             "message": err.Error(),
35         })
36         return
37     }
38
39     c.JSON(http.StatusCreated, Product)
40 }
```

Create Product Endpoint

Jika kita melihat pada gambar diatas, pada line 15 kita menggunakan method *MustGet* yang merupakan sebuah method dari framework **Gin**.

Kita menggunakan method *MustGet* untuk mendapatkan claim dari token yang telah disimpan oleh middleware *authentication* yang telah kita buat sebelumnya.

Data apapun yang kita dapatkan melalui method *MustGet* akan berubah tipe datanya menjadi *empty interface*, maka dari itu kita perlu mengcasting kembali data yang di dapatkan dari method *MustGet* menjadi tipe data semulanya.

Kemudian pada line 27, kita memasukkan *id* dari user yang didapatkan melalui method *MustGet* kedalam struct *Product* sebelum kita membuat data product baru.

```
13 func CreateProduct(c *gin.Context) {
14     db := database.GetDB()
15     userData := c.MustGet("userData").(jwt.MapClaims)
16     contentType := helpers.GetContentType(c)
17
18     Product := models.Product{}
19     userID := uint(userData["id"].(float64))
20
21     if contentType == appJSON {
22         c.ShouldBindJSON(&Product)
23     } else {
24         c.ShouldBind(&Product)
25     }
26
27     Product.UserID = userID
28
29     err := db.Debug().Create(&Product).Error
30
31     if err != nil {
32         c.JSON(http.StatusBadRequest, gin.H{
33             "err": "Bad Request",
34             "message": err.Error(),
35         })
36         return
37     }
38
39     c.JSON(http.StatusCreated, Product)
40 }
```

Create Product Route

Sekarang kita akan membuat routing untuk product sekaligus menempatkan middleware nya.

```
10 func StartApp() *gin.Engine {
11     r := gin.Default()
12
13     userRouter := r.Group("/users")
14     {
15         userRouter.POST("/register", controllers.UserRegister)
16
17         userRouter.POST("/login", controllers.UserLogin)
18     }
19
20
21     productRouter := r.Group("/products")
22     {
23         productRouter.Use(middlewares.Authentication())
24         productRouter.POST("/", controllers.CreateProduct)
25     }
26
27     return r
28 }
```


Create Product Route

Bisa kita lihat pada line 23, ketika kita ingin mengimplementasikan sebuah middleware pada framework **Gin**, maka kita dapat menggunakan method *Use* dari framework **Gin**.

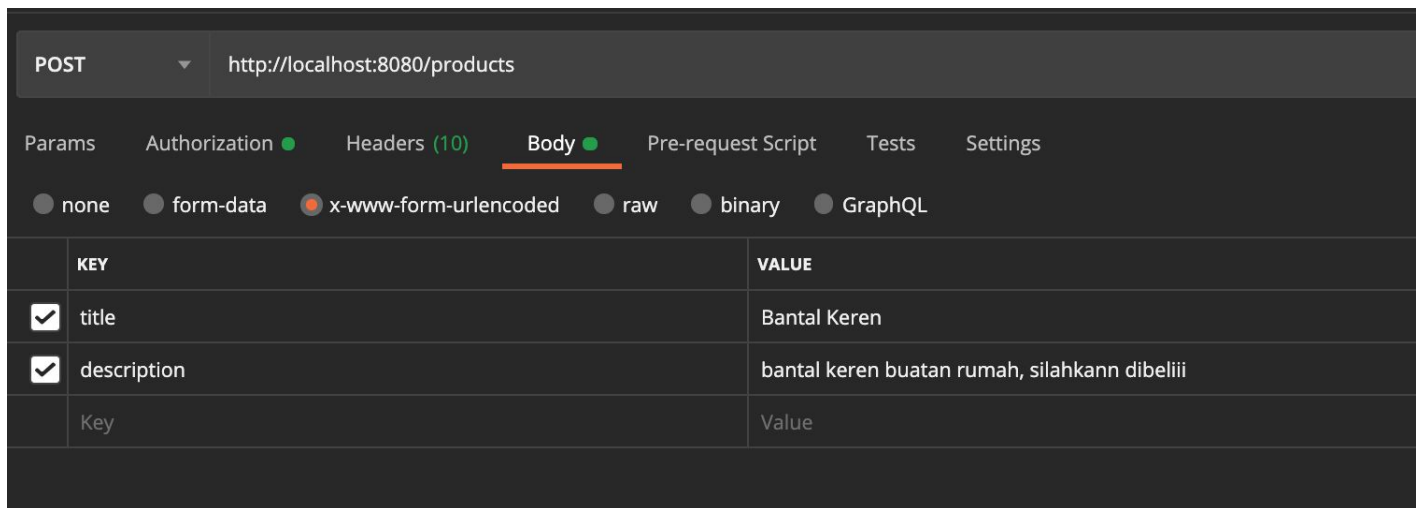
Pada line 23, kita memanggil middleware *Authentication* sebelum pada baris pertama dari routing product.

Hal tersebut kita lakukan karena kita ingin sebelum client memasuki seluruh endpoint dari product, maka harus melewati proses autentikasi atau pengecekan token terlebih dahulu.

```
10 func StartApp() *gin.Engine {
11     r := gin.Default()
12
13     userRouter := r.Group("/users")
14     {
15         userRouter.POST("/register", controllers.UserRegister)
16
17         userRouter.POST("/login", controllers.UserLogin)
18     }
19
20
21     productRouter := r.Group("/products")
22     {
23         productRouter.Use(middlewares.Authentication())
24         productRouter.POST("/", controllers.CreateProduct)
25     }
26
27     return r
28 }
```

Test Create Product Endpoint With Postman

Mari kita buat sebuah request untuk mencoba endpoint create product yang baru saja kita buat. Buatlah sebuah request seperti pada gambar dibawah ini.



POST http://localhost:8080/products

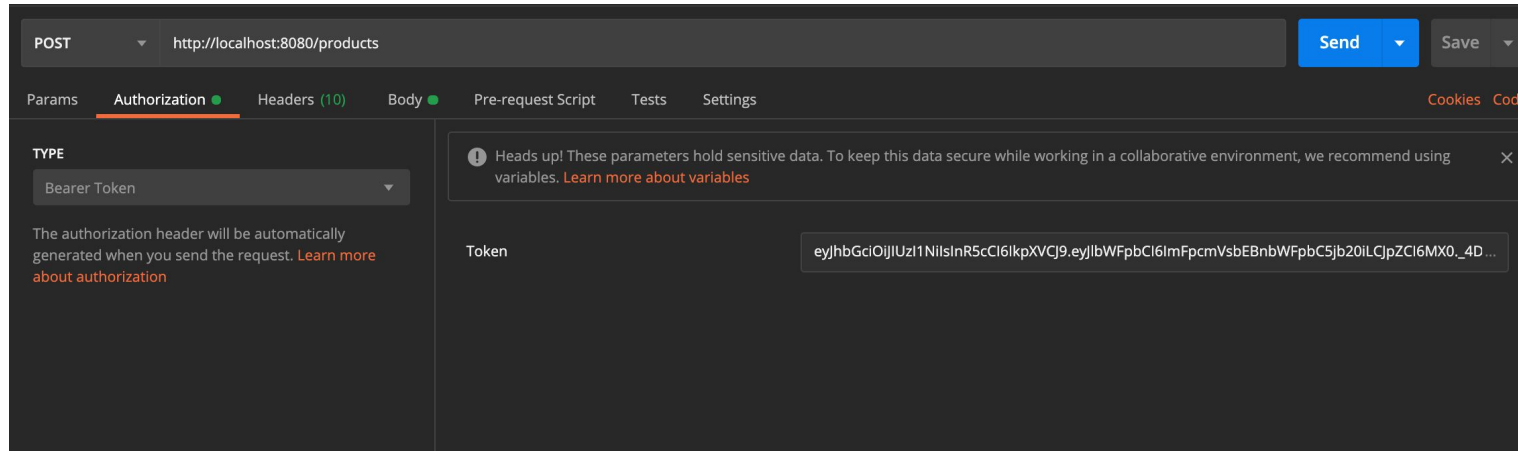
Params Authorization Headers (10) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	KEY	VALUE
<input checked="" type="checkbox"/>	title	Bantal Keren
<input checked="" type="checkbox"/>	description	bantal keren buatan rumah, silahkann dibeliii
	Key	Value

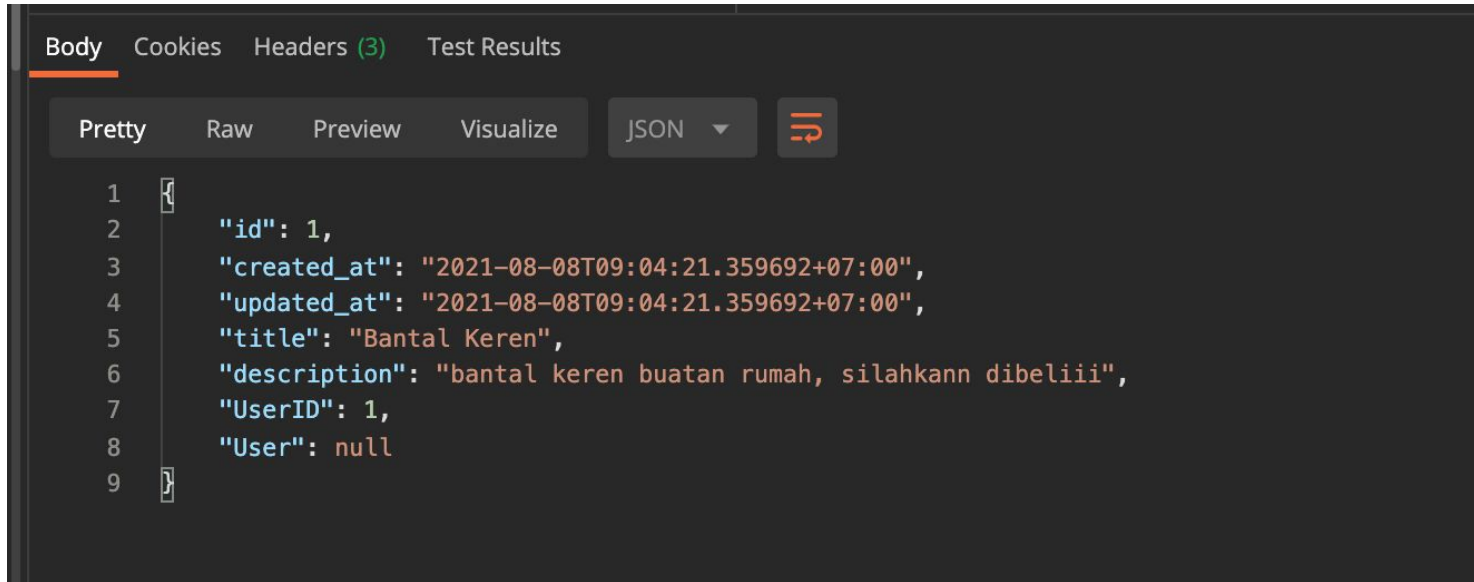
Test Create Product Endpoint With Postman

Lalu jangan lupa klik bagian Authorization, lalu pilih bagian type nya sebagai Bearer Token. Kemudian jangan lupa untuk meletakkan token yang kita dapatkan dari hasil user login pada bagian Token. Contohnya seperti pada gambar dibawah ini.



Test Create Product Product Endpoint With Postman

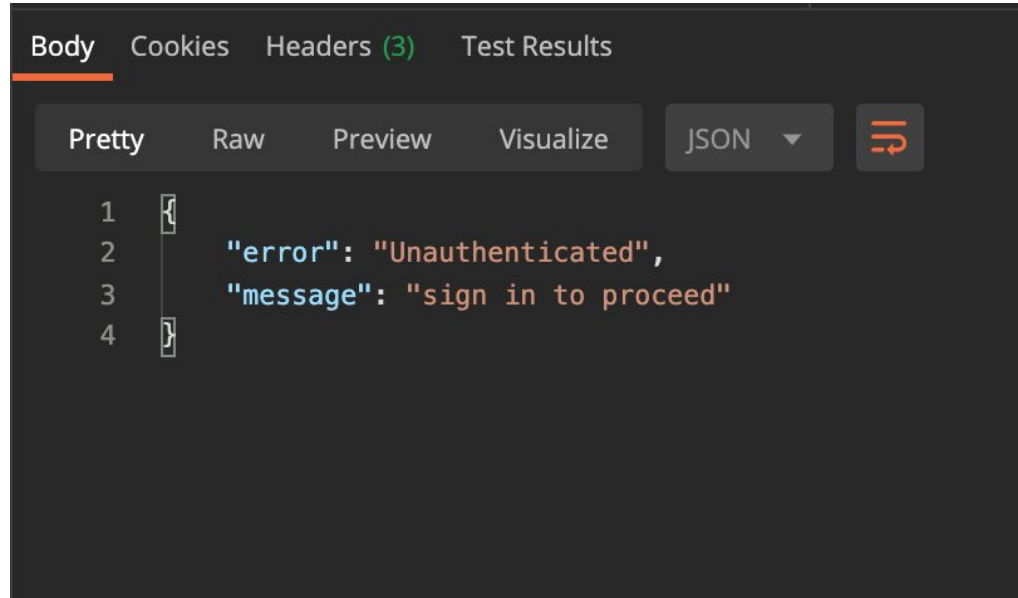
Kemudian kirimkanlah request nya, maka kita akan mendapatkan response seperti pada gambar dibawah ini.

A screenshot of the Postman application showing a successful JSON response. The interface includes tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results'. The 'Body' tab is active, displaying a JSON object in 'Pretty' format. The JSON contains fields for 'id', 'created_at', 'updated_at', 'title', 'description', 'UserID', and 'User'.

```
1 {  
2   "id": 1,  
3   "created_at": "2021-08-08T09:04:21.359692+07:00",  
4   "updated_at": "2021-08-08T09:04:21.359692+07:00",  
5   "title": "Bantal Keren",  
6   "description": "bantal keren buatan rumah, silahkann dibeliii",  
7   "UserID": 1,  
8   "User": null  
9 }
```

Test Create Product Product Endpoint With Postman

Jika teman-teman mencoba untuk tidak mengirimkan token, maka akan mendapatkan response error seperti pada gambar dibawah ini.



The image shows a screenshot of the Postman application interface. The 'Body' tab is selected, and the response is displayed in 'Pretty' format. The response is a JSON object with an 'error' field set to 'Unauthenticated' and a 'message' field set to 'sign in to proceed'.

```
1 {  
2   "error": "Unauthenticated",  
3   "message": "sign in to proceed"  
4 }
```

Create Authorization Middleware

Sekarang kita akan membuat middleware untuk proses autotorisasi. Proses autorisasi ini kita buat agar user hanya dapat mengupdate atau menghapus data product miliki nya sendiri.

Buatlah sebuah file dengan nama **authorization.go** pada folder **middlewares**.

Kemudian pada file tersebut, buatlah sebuah function dengan nama *Authorization* seperti pada gambar di halalaman berikutnya.

```
13 func ProductAuthorization() gin.HandlerFunc {
14     return func(c *gin.Context) {
15         db := database.GetDB()
16         productId, err := strconv.Atoi(c.Param("productId"))
17         if err != nil {
18             c.AbortWithStatusJSON(http.StatusBadRequest, gin.H{
19                 "error": "Bad Request",
20                 "message": "invalid parameter",
21             })
22             return
23         }
24         userData := c.MustGet("userData").(jwt.MapClaims)
25         userID := uint(userData["id"].(float64))
26         Product := models.Product{}
27
28         err = db.Select("user_id").First(&Product, uint(productId)).Error
29     }
```

```
30     if err != nil {
31         c.AbortWithStatusJSON(http.StatusNotFound, gin.H{
32             "error": "Data Not Found",
33             "message": "data doesn't exist",
34         })
35         return
36     }
37
38     if Product.UserID != userID {
39         c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{
40             "error": "Unauthorized",
41             "message": "you are not allowed to access this data",
42         })
43         return
44     }
45
46     c.Next()
47 }
48 }
```

Create Authorization Middleware

Mari kita bedah maksud dari syntax pada gambar di halaman sebelumnya:

- Pada line 16, kita mencoba untuk mendapatkan route parameter berupa *productId*, karena pastinya untuk proses update dan delete product memerlukan route parameter.
- Pada line 24, kita mencoba untuk claim token yang telah disimpan oleh proses autentikasi. Perlu diingat disini bahwa setiap proses autorisasi harus melewati proses autentikasi terlebih dahulu, bukan sebaliknya.
- Pada line 28, kita mencoba untuk mendapatkan data berdasarkan id product yang didapatkan dari route parameter berupa *productId*.
- Pada line 30 - 36, kita melakukan pengecekan dengan alur jika productnya tidak ada maka kita akan melempar error berupa data not found (**404**).
- Pada line 38-44, kita melakukan pengecekan dengan alur jika product yang didapatkan memiliki nilai dari field *UserId* yang sama dengan *id* user yang didapatkan dari claim token, maka proses dapat dilanjut ke endpoint berikutnya. Namun jika tidak sama, maka kita akan melempar error berupa *Unauthorized* dengan status **403**.

Update Product Endpoint

Pada file **product.go** di dalam folder **controllers**, buat suatu function dengan nama *UpdateProduct* seperti pada gambar dibawah ini.

```
43 func UpdateProduct(c *gin.Context) {
44     db := database.GetDB()
45     userData := c.MustGet("userData").(jwt.MapClaims)
46     contentType := helpers.GetContentType(c)
47     Product := models.Product{}
48
49     productId, _ := strconv.Atoi(c.Param("productId"))
50     userID := uint(userData["id"].(float64))
51
52     if contentType == appJSON {
53         c.ShouldBindJSON(&Product)
54     } else {
55         c.ShouldBind(&Product)
56     }
57
58     Product.UserID = userID
59     Product.ID = uint(productId)
60
61     err := db.Model(&Product).Where("id = ?", productId).Updates(models.Product{Title: Product.Title, Description: Product.Description}).Error
62
63     if err != nil {
64         c.JSON(http.StatusBadRequest, gin.H{
65             "err": "Bad Request",
66             "message": err.Error(),
67         })
68         return
69     }
70
71     c.JSON(http.StatusOK, Product)
72 }
```


Update Product Router

Sekarang pada file **router.go**, tambahkan satu routing yang akan mengarah pada endpoint update product. Caranya seperti pada gambar dibawah ini.

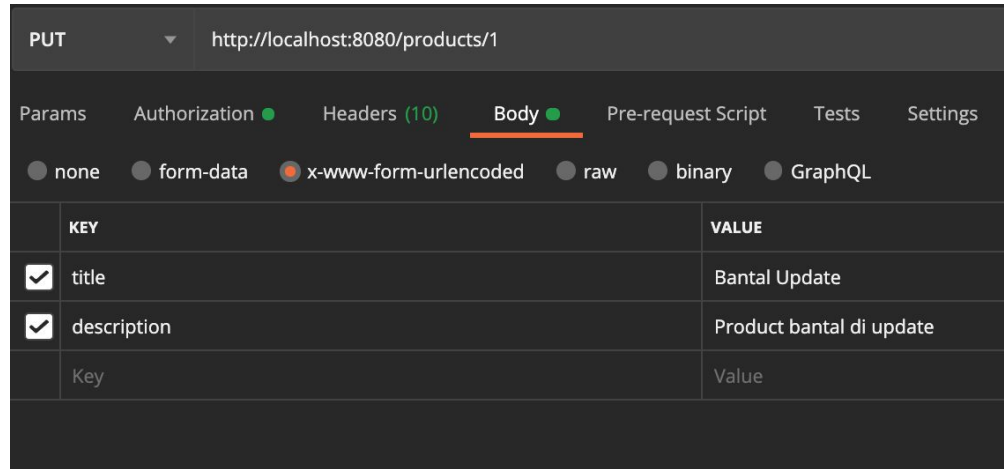
```
21 productRouter := r.Group("/products")
22 {
23     productRouter.Use(middlewares.Authentication())
24     productRouter.POST("/", controllers.CreateProduct)
25
26     productRouter.PUT("/:productId", middlewares.ProductAuthorization(), controllers.UpdateProduct)
27 }
28
29 return r
30 }
```

Jika kita perhatikan pada line 26 diatas, kita menempatkan middleware *PhotoAuthorization* di antara route path dengan endpointnya.

Hal tersebut dapat kita lakukan ketika kita hanya membutuhkan suatu middleware untuk suatu endpoint tertentu.

Test Update Product Endpoint With Postman

Mari kita buat sebuah request untuk mencoba endpoint update product yang baru saja kita buat. Jangan lupa untuk menaruh token pada bagian **Authorization**. Buatlah sebuah request seperti pada gambar dibawah ini.



PUT http://localhost:8080/products/1

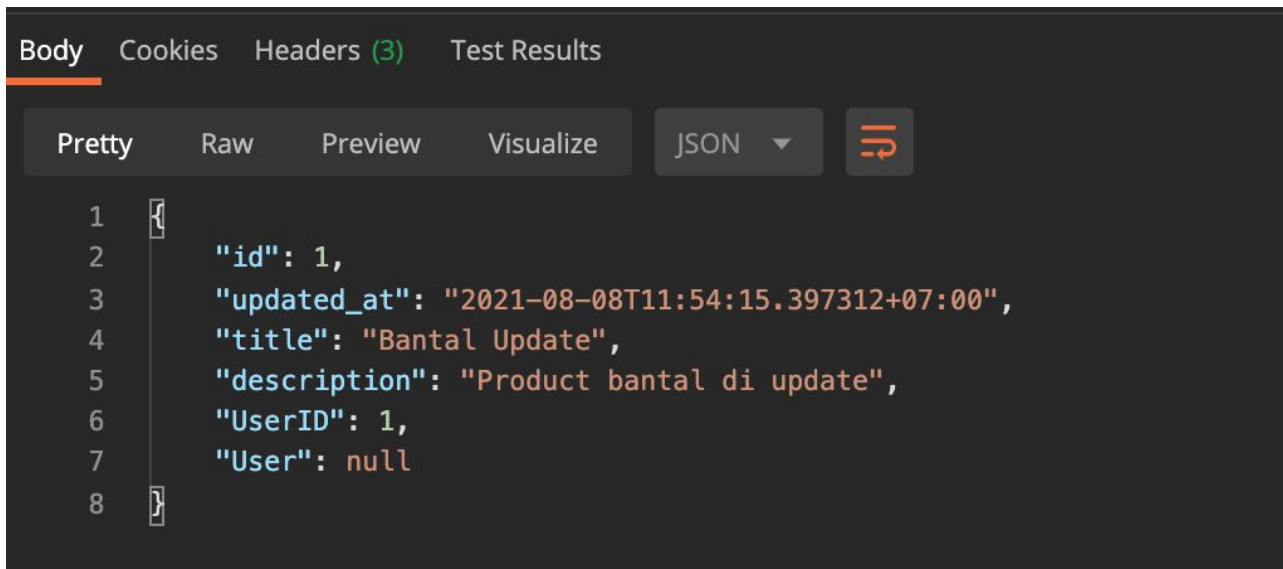
Params Authorization Headers (10) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	KEY	VALUE
<input checked="" type="checkbox"/>	title	Bantal Update
<input checked="" type="checkbox"/>	description	Product bantal di update
	Key	Value

Test Update Product Endpoint With Postman

Maka kita akan mendapatkan response seperti pada gambar dibawah ini.

A screenshot of the Postman application showing a JSON response in the 'Body' tab. The response is a JSON object with fields for id, updated_at, title, description, UserID, and User. The 'User' field is null. The interface includes tabs for Body, Cookies, Headers (3), and Test Results. Below the tabs are buttons for Pretty, Raw, Preview, and Visualize, along with a JSON dropdown and a refresh icon. The JSON is displayed in a dark-themed editor with line numbers 1 through 8.

```
1 {  
2   "id": 1,  
3   "updated_at": "2021-08-08T11:54:15.397312+07:00",  
4   "title": "Bantal Update",  
5   "description": "Product bantal di update",  
6   "UserID": 1,  
7   "User": null  
8 }
```