HACKTIV8

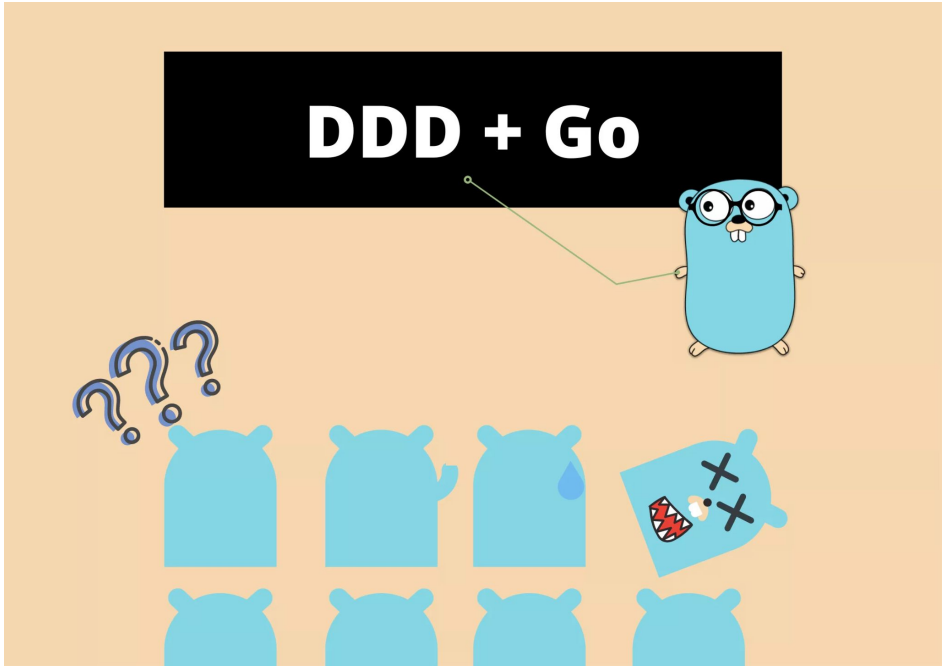
Go Programming Microservice Into The Specialization

Sesi 2 : DDD, Go Micro, Web Services & Web Server Go Programming

HACKTIV8

DDD
(Domain Driven Design)

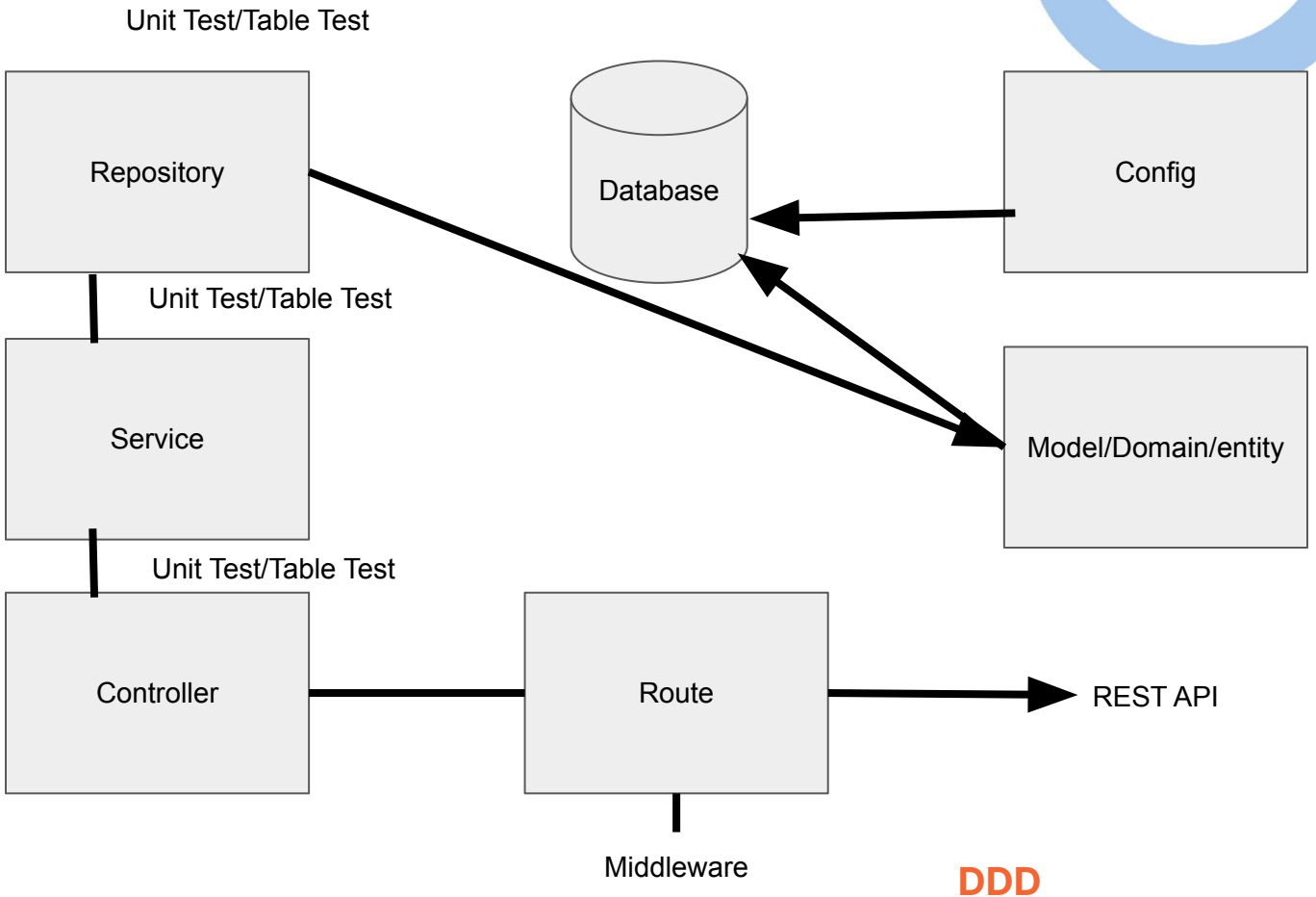
Domain Driven Design



Domain-driven design (DDD) atau kalau kita diterjemahkan secara bebas sebagai desain berbasis domain, seperti yang dicetuskan oleh **Eric Evans** pada bukunya yang berjudul sama,memperkenalkan beberapa konsep penting yang diperlukan untuk membuat dan membangun microservice berbasis event.

Konsep ini diimplementasikan dengan tujuan agar struktur folder project, serta struktur coding tertata bersih dan rapih serta memiliki basis flow yang berbasis object dan memiliki pendekatan agile.

Slide berikutnya merupakan refrensi atau contoh penggunaan artitektur DDD pada bentuk pembuatan folder dan flownya

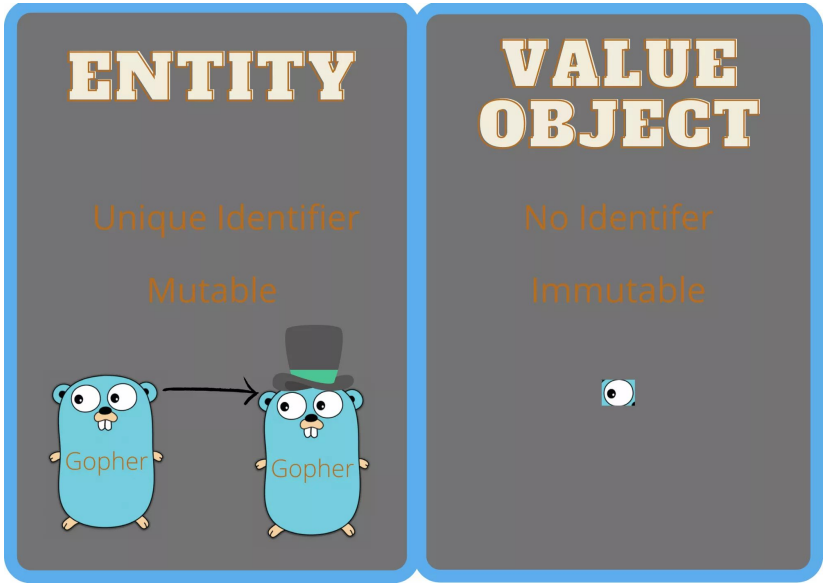


HACKTIV8

DDD

Domain Driven Design

Domain Driven Design

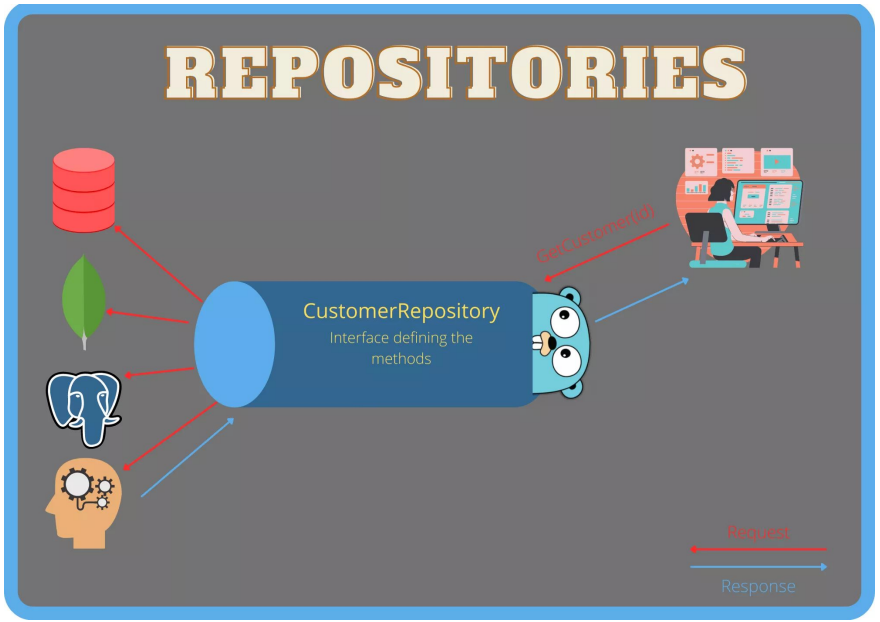


DDD

Domain

- Entity
- Config
- Driver

Domain Driven Design

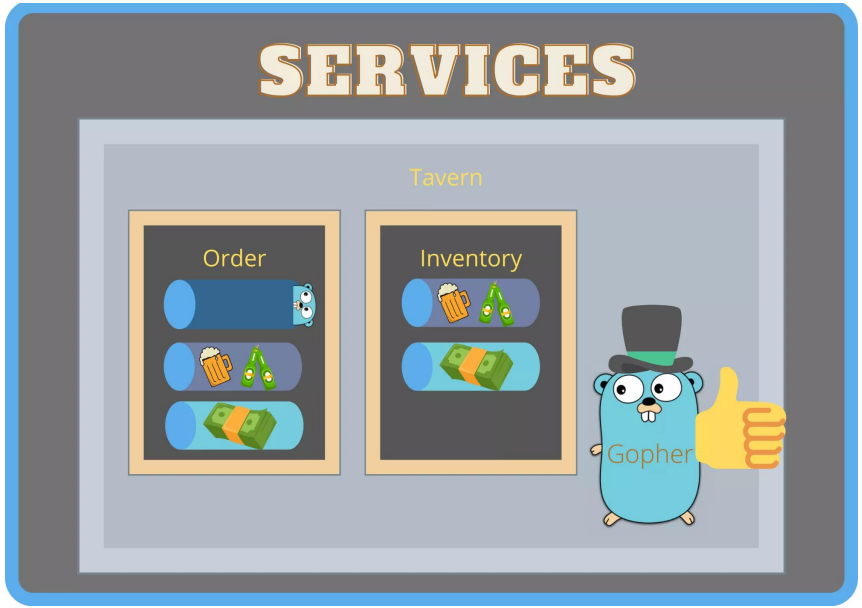


DDD

Repositories

- Query
- Third Parrties
- Method Collection

Domain Driven Design

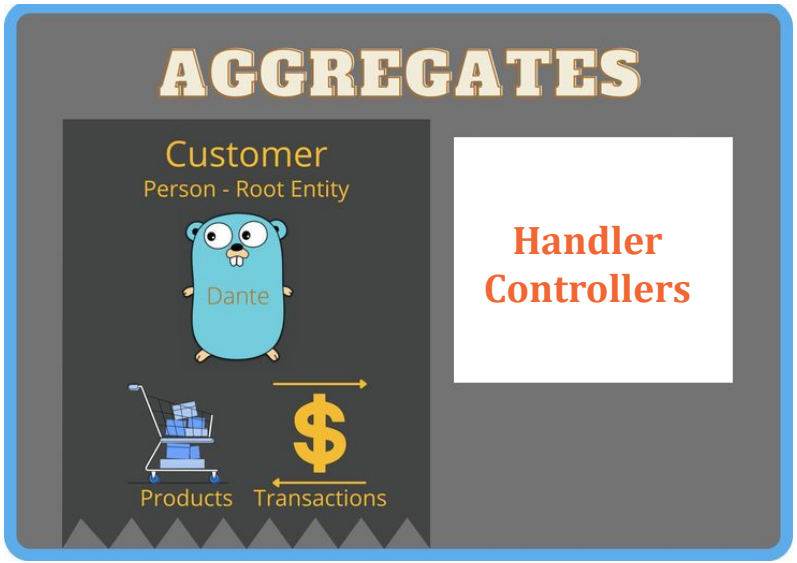


DDD

Services

- Logic backend
- Core flow backend
- Middleware
- Helpers service for logic

Domain Driven Design



DDD

Aggregates/Interface

- Controllers/Handler
- Routes
- Web Server

Web Server

HACKTIV8

Web Server

Go telah menyediakan sebuah package bernama *net/http* untuk berbagai macam keperluan dalam membuat sebuah aplikasi berbasis web seperti contohnya routing, templating, web server dan lain-lain. Dan pada materi kali ini kita akan belajar bagaimana cara membuat web server beserta routingnya pada bahasa Go.

Sekarang buatlah satu file dengan nama **web.go** dan isilah dengan syntax seperti pada gambar di sebelah kanan. Pada line 8, kita mempunyai sebuah variable global bernama *PORT* yang dimana kita menyimpan nilai port locahost yang mengarah pada **localhost:8080**. Lalu pada line 11, kita memakai function *HandleFunc* yang berasal dari package *http*. Function *HandleFunc* digunakan untuk keperluan routing kita, yang dimana function tersebut menerima 2 parameter. Parameter pertama digunakan untuk mendefinisikan path routingnya, sedangkan parameter kedua menerima sebuah function dengan 2 parameter yaitu http.ResponseWriter dan pointer http.Request

```
1  package main
2
3  import (
4    "fmt"
5    "net/http"
6  )
7
8  var PORT = ":8080"
9
10 func main() {
11   http.HandleFunc("/", greet)
12
13   http.ListenAndServe(PORT, nil)
14 }
15
16 func greet(w http.ResponseWriter, r *http.Request) {
17   msg := "Hello World"
18   fmt.Fprint(w, msg)
19 }
```

Web Server

http.ResponseWriter adalah sebuah interface dengan berbagai method yang digunakan untuk mengirim response balik kepada client yang mengirimkan request. Kemudian *http.Request* adalah sebuah struct yang digunakan untuk mendapat berbagai macam data request seperti form value, headers, url parameter dan lain-lain.

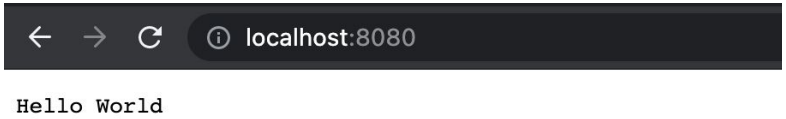
Kemudian pada line 13, kita menggunakan function *ListenAndServe* untuk menjalankan server aplikasi. Function *ListenAndServe* menerima 2 parameter yaitu keterangan port yang kita pakai, dan http.Handler yang merupakan sebuah *interface*. Namun karena kita tidak menggunakan *http.Handler*, maka kita cukup memberikan zero value dari tipe data *interface* yaitu *nil*.

Kemudian pada line 16 - 19, kita membuat function *greet* yang dimana kita gunakan untuk mengirim response berupa tulisan **“Hello World”** kepada client. Lalu untuk mengirim response nya kita menggunakan function *fmt.Fprint*.

```
1  package main
2
3  import (
4    "fmt"
5    "net/http"
6  )
7
8  var PORT = ":8080"
9
10 func main() {
11   http.HandleFunc("/", greet)
12
13   http.ListenAndServe(PORT, nil)
14 }
15
16 func greet(w http.ResponseWriter, r *http.Request) {
17   msg := "Hello World"
18   fmt.Fprint(w, msg)
19 }
```

Web Server

Sekarang jalankan server aplikasi kita pada terminal dengan perintah **go run web.go**. Kemudian bukalah browser dan arahkan urk kepada http://localhost:8080/. Dan kita akan melihat sebuah tulisan berupa **Hello World** seperti pada gambar di sebelah kanan.



API

Sekarang kita akan mencoba untuk membuat sebuah API yang sangat simple. Yang dimana kita akan menggunakan 2 method saja berupa GET dan POST yang dimana akan kita gunakan untuk mendapatkan data employee dan membuat data employee baru. Untuk itu kita akan membuat data-data employee itu terlebih dahulu dengan menyimpannya didalam sebuah variable dengan tipe data *slice* yang berisikan tipe data struct *Employee*.

Maka dari itu, buat lah sebuah file dengan nama **api.go**, lalu kemudian ikutilah syntax-syntax seperti pada gambar disebelah kanan.

```go
package main

import (
  "encoding/json"
  "fmt"
  "net/http"
  "strconv"
)

type Employee struct {
  ID       int
  Name     string
  Age      int
  Division string
}

var employees = []Employee{
  {ID: 1, Name: "Airell", Age: 23, Division: "IT"},
  {ID: 2, Name: "Nanda", Age: 23, Division: "Finance"},
  {ID: 3, Name: "Mailo", Age: 20, Division: "IT"},
}

var PORT = ":8080"

```

API

Kemudian ikutilah syntax seperti pada gambar disebelah kanan untuk function *main*, dan juga untuk function *getEmployees*.

Pada line 33, kita menggunakan method *Header* dari interface *http.ResponseWriter* yang kemudian di chaining dengan method *Set*. Hal ini kita lakukan untuk menentukan bentuk dari data response yang ingin kita kirimkan kepada client. Kita perlu mengatur *Content-Type* menjadi *application/json* dalam method *Set*.

Pada line 35 kita melakukan pengecekan *method*. Karena function *getEmployees* kita gunakan untuk mendapatkan data-data empoyee dan pada umumnya untuk mendapatkan data dari server menggunakan method *GET*. Pada line 36, kita mengkonversi data *employees* menjadi data berbentuk JSON untuk dikirimkan kepada client dengan menggunakan method *NewEncoder* yang berasal dari package *json*, yang kemudian kita chaining dengan method *Encode* untuk mengkonversi datanya menjadi bentuk JSON.

```
25  func main() {
26    http.HandleFunc("/employees", getEmployees)
27
28    fmt.Println("Application is listening on port", PORT)
29    http.ListenAndServe(PORT, nil)
30  }
31
32  func getEmployees(w http.ResponseWriter, r *http.Request) {
33    w.Header().Set("Content-Type", "application/json")
34
35    if r.Method == "GET" {
36      json.NewEncoder(w).Encode(employees)
37      return
38    }
39
40    http.Error(w, "Invalid method", http.StatusBadRequest)
41  }
```

API

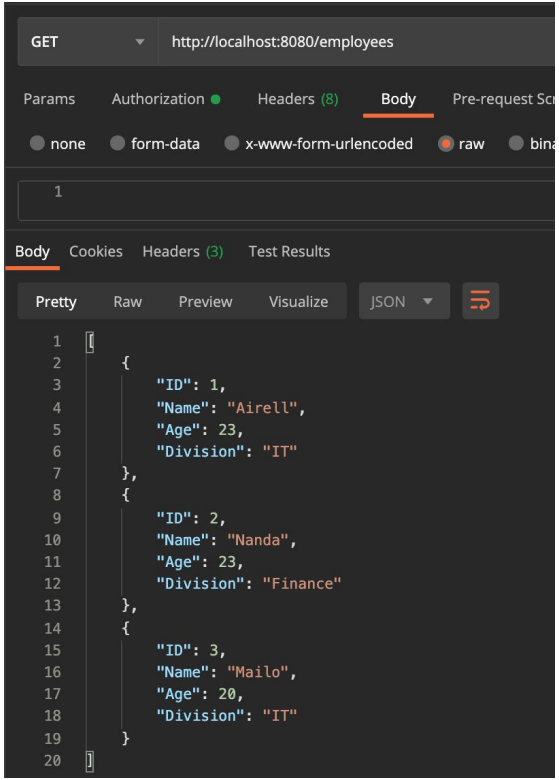
Kemudian jika method yang dikirimkan oleh client bukan method *GET*, maka kita akan mengirimkan response error dengan menggunakan function Error dari package *http*. Lalu *http.StatusBadRequest* merupakan sebuah konstanta dari package *http.StatusBadRequest* yang merepresentasikan status *400*.

```
25  func main() {
26    http.HandleFunc("/employees", getEmployees)
27
28    fmt.Println("Application is listening on port", PORT)
29    http.ListenAndServe(PORT, nil)
30  }
31
32  func getEmployees(w http.ResponseWriter, r *http.Request) {
33    w.Header().Set("Content-Type", "application/json")
34
35    if r.Method == "GET" {
36      json.NewEncoder(w).Encode(employees)
37      return
38    }
39
40    http.Error(w, "Invalid method", http.StatusBadRequest)
41  }
```

API

Sekarang kita akan mencoba untuk mengirimkan request kepada server kita dengan menggunakan *Postman*. Jalankan server aplikasi kita dengan perintah **go run api.go**, dan buat request pada postman yang mengarah pada URL http://localhost:8080/employees dengan method GET.

Maka setelah itu kita akan mendapatkan data response berbentuk JSON seperti pada gambar di sebelah kanan.



HACKTIV8

API

Sebelumnya kita telah menggunakan method *GET* untuk mendapatkan seluruh data employee. Unutk sekarang kita akan mencoba menerakan method *POST* yang dimana kita gunakan untuk menambah data employee baru. Maka dari itu buatlah sebuah function baru dengan nama createEmployee seperti pada gambar di sebelah kanan.

Pada line 49 - 51, kita mencoba untuk mendapatkan nilai input form dari client dengan menggunakan method *FormValue* yang berasal dari struct *http.Request*. Kemudian pada line 53 kita mengkonversi input *age* dari tipe data string menjadi tipe data *int* karena seluruh nilai input yang kita dapat dari client akan memiliki tipe data *string*, sedangkan field *age* pada struct *Employee* memiliki tipe data *int*. Setelah itu pada line 60, kita membuat variable *newEmployee* yang akan menampung nilai-nilai input dari client yang kemudian akan kita append atau masukkan kedalam slice *employees*.

```
45  func createEmployee(w http.ResponseWriter, r *http.Request) {
46    w.Header().Set("Content-Type", "application/json")
47
48    if r.Method == "POST" {
49      name := r.FormValue("name")
50      age := r.FormValue("age")
51      divison := r.FormValue("division")
52
53      convertAge, err := strconv.Atoi(age)
54
55      if err != nil {
56        http.Error(w, "Invalid age", http.StatusBadRequest)
57        return
58      }
59
60      newEmployee := Employee{
61        ID:       len(employees) + 1,
62        Name:     name,
63        Age:      convertAge,
64        Division: divison,
65      }
66
67      employees = append(employees, newEmployee)
68
69      json.NewEncoder(w).Encode(newEmployee)
70      return
71    }
72
73    http.Error(w, "Invalid method", http.StatusBadRequest)
74  }
```

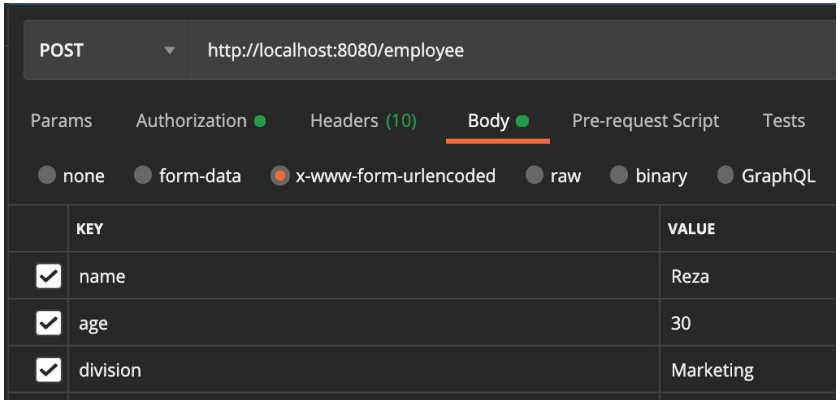
API

Sebelumnya kita mencoba untuk mengirimkan request menggunakan Postman, kita perlu meregistrasikan function ***createEmployee*** kedalam routingan kita. Kemudian jalankan ulang server aplikasi kita, lalu buatlah nilai-nilai input pada Postman seperti pada gambar kedua.

```go
func main() {
  http.HandleFunc("/employees", getEmployees)

  http.HandleFunc("/employee", createEmployee)

  fmt.Println("Application is listening on port", PORT)
  http.ListenAndServe(PORT, nil)
}
```



API

Sekarang kirimkan request dari Postman yang mengarah pada URL http://localhost:8080/employee dengan method *POST.* Maka kita akan mendapatkan data response seperti pada gambar pertama disebelah kanan.

Kemudian jika kita kirimkan request kembali yang mengarah pada URL http://localhost:8080/employees dengan method *GET*, maka kita akan dapat melihat data employee yang baru saja kita buat sudah ada di dalam kumpulan data employee nya seperti pada gambar kedua.

API

Kita juga dapat mengirimkan data employee dengan dengan template html. Bahasa Go telah menyediakan suatu package bernama *html/template* untuk melakukannya.

Sekarang buatlah satu file html dengan nama **template.html** yang masih berada pada direktori yang sama dengan file **api.go** yang kita buat sebelumnya. Kemudian ikutilah syntax seperti pada gambar di sebelah kanan untuk file htmlnya.

Jika kita perhatikan, terdapat tanda seperti **{{.}}**. Ketika kita ingin menampilkan data pada file html, maka kita perlu menyelipkan data tersebut kedalam dua tanda kurung **{{}}**, lalu menempatkan tand titik didalamnya seperti yang kita lihat pada gambar di sebelah kanan. Tanda titik **.** yang berada di dalam 2 tanda kurung tersebut merepresentasikan data yang ingin kita tampilkan.

```
<html>
  <head>
    <title>HTML</title>
  </head>
  <body>
    <div>
      {{.}}
    </div>
    </body>
</html>
```

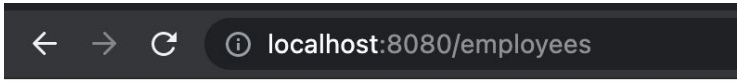
API

Kemudian kita akan mengubah sedikit isi dari function *getEmployees* yang sudah kita buat sebelumnya menjadi seperti pada gambar di sebelah kanan.

```
35  func getEmployees(w http.ResponseWriter, r *http.Request) {
36
37    if r.Method == "GET" {
38      tpl, err := template.ParseFiles("template.html")
39
40      if err != nil {
41        http.Error(w, err.Error(), http.StatusInternalServerError)
42        return
43      }
44
45      tpl.Execute(w, employees)
46      return
47    }
48
49    http.Error(w, "Invalid method", http.StatusBadRequest)
50  }
```

Pada line 38, kita menggunakan function *template.ParseFiles* yang berasal dari package *html/template* yang digunakan untuk mem-parsing file html kita. Karena file html yang baru kita buat bernama **template.html,** maka dari itu kita meletakkan nama file html kita sebagai argumen function *template.ParseFiles*.

Kemudian jika sudah berhasil di parsing oleh function *template.ParseFiles*, maka function tersebut akan mengembaikan suatu tipe data struct *template.*



[{1 Airell 23 IT} {2 Nanda 23 Finance} {3 Mailo 20 IT}]

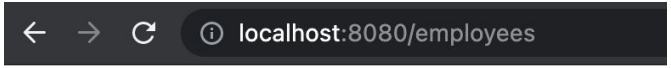
Template yang mempunyai sebuah method bernama *Execute* yang digunakan untuk memberikan response kepada client berupa template html. Parameter kedua dari method *Execute* digunakan untuk mengirimkan data yang ingin kita tampilkan pada file html kita.

API

Jika kita ingin dapat melooping data employee nya, maka kita dapat melakukannya dengan menggunakan range loop seperti pada gambar di sebelah kanan. Formata pernulisa range loop nya adalah **{{range $namaVariable := .}}**, yang dimana cara penulisan nama variablenya masih sama seperti biasa namun harus diawali dengan tanda dolar **$**. Lalu setelah itu kita juga perlu mengakhiri looping nya dengan keyword *end* yang juga perlu dimasukkan kedalam 2 tanda kurung **{{end}}**.

Sekarang arahkan browser kita kembali kepada URL http://localhost:8080/employees, dan kita akan melihat hasilnya seperti pada gambar kedua.

```
<html>
  <head>
    <title>HTML</title>
  </head>
  <body>
    <div>
      <ul>
        {{range $value := .}}
        <li>
          ID: {{$value.ID}}
          Name: {{$value.Name}},
          Age: {{$value.Age}},
          Division: {{$value.Division}}
        </li>
        {{end}}
      </ul>
    </div>
    </body>
</html>
```



- ID: 1 Name: Airell, Age: 23, Division: IT
- ID: 2 Name: Nanda, Age: 23, Division: Finance
- ID: 3 Name: Mailo, Age: 20, Division: IT

Gin Framework

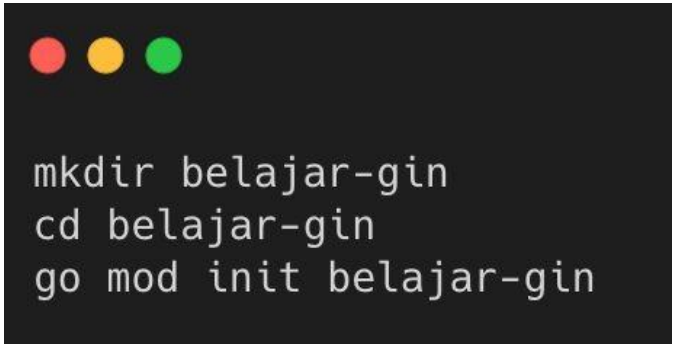
HACKTIV8

Introduction

Gin adalah sebauh framework untuk bahasa Go yang digunakan untuk keperluan http routing. Dengan menggunakan **Gin**, maka akan sangat mempermudah kita dalam membuat **Rest API** yang sudah pasti memerlukan fitur routing.

Kali ini kita akan membuat sebuah Rest API sederhana yang akan melakukan CRUD data mobil.

Untuk itu, maka buatlah sebuah folder dengan nama **belajar-gin**, dan jalankan perintah go mod init **belajar-gin**
pada folder tersebut.



HACKTIV8

Introduction

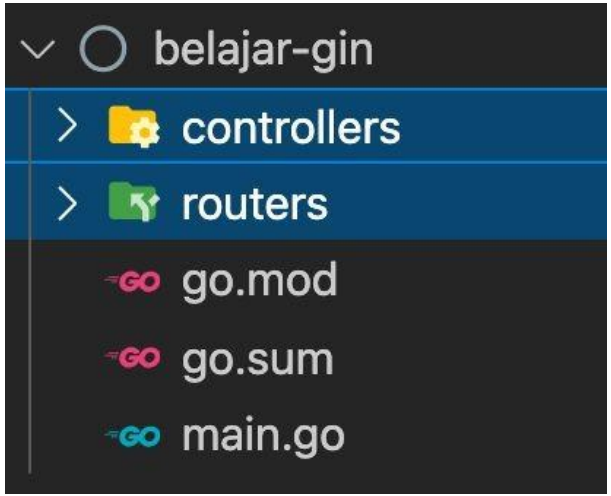
Kemudian jalankan perintah **go get -u github.com/gin-gonic/gin** pada terminal untuk menginstall framework Gin.

HACKTIV8

Setting Environment

Buat suatu file bernama **main.go** pada root direktori. File **main.go** itu nantinya akan menjadi entry point dari servernya.

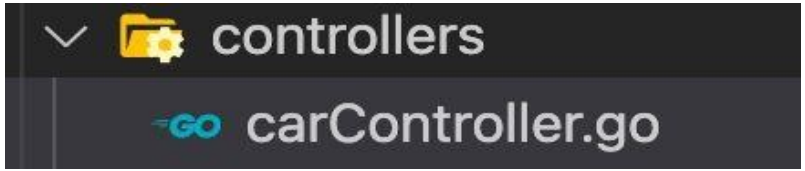
Kemudian buatlah 2 folder dengan nama **routers** dan juga **controllers**. Folder **routers** akan menjadi tempat kita dalam menaruh konfigurasi dari routingnya, sedangkan **controllers** akan menjadi tempat kita untuk menaruh endpoint-endpoint yang kita perlukan.



Setting Controller

Pada folder **controllers**, buatlah sebuah file dengan nama **carController.go**.

Kemudian pada file tersebut, buat lah sebuah struct dengan nama **Car** yang akan kita jadikan sebagai struktur data dari data-data mobilnya, dan kita juga memerlukan sebuah variable global dengan tipe data **list** yang akan menampung seluruh data mobilnya.




```go
type Car struct {
  CarID string `json:"car_id"`
  Brand string `json:"brand"`
  Model string `json:"model"`
  Price int    `json:"price"`
}

var CarDatas = []Car{}
```

Make Car Controller

Sekarang kita akan membuat sebuah endpoint unutk keperluan create data atau membuat data mobil baru.

Untuk itu pada file **carController.go**, kita perlu mengimport dulu framework **Gin** yang telah kita install dengan cara seperti pada gambar dibawah ini.

Kita juga perlu mengimport package **net/http** untuk penentuan status code dan package **fmt** untuk memformat data string.

```
1  package controllers
2
3  import (
4    "fmt"
5    "net/http"
6
7    "github.com/gin-gonic/gin"
8  )
```

HACKTIV8

Create Car

Kemudian buatlah sebuah function dengan nama **CreateCar** yang menerima satu parameter dengan tipe data ***gin.Context**.

Function **CreateCar** akan menjadi endpoint untuk proses **create data**. Function ini perlu menerima sebuah parameter dengan tipe data ***gin.Context** yang merupakan sebuah tipe data yang telah disediakan oleh framework **Gin**.

***gin.Context** mempunyai berbagai macam **method** yang dapat kita gunakan untuk mendapat request body dari client, mengirim response dan lain-lain.

```
19  func CreateCar(ctx *gin.Context) {
20
21  }
```

Create Car

Ikutilah kode-kode pada gambar di sebelah kanan untuk melengkapi proses dari membuat data mobil baru.

```
19  func CreateCar(ctx *gin.Context) {
20    var newCar Car
21
22    if err := ctx.ShouldBindJSON(&newCar); err != nil {
23      ctx.AbortWithError(http.StatusBadRequest, err)
24      return
25    }
26    newCar.CarID = fmt.Sprintf("c%d", len(CarDatas)+1)
27    CarDatas = append(CarDatas, newCar)
28
29    ctx.JSON(http.StatusCreated, gin.H{
30      "car": newCar,
31    })
32  }
```

ShouldBindJSON pada baris **22** merupakan sebuah method dari tipe data ***gin.Context** yang digunakan untuk mem-binding data **JSON** yang kirimkan oleh client sebagai request body kepada server. Method **ShouldBindJSON**
menerima sebuah parameter yang dimana kita perlu meletakkan pointer dari variable yang akan menampung hasil dari data binding tersebut.

Method **ShouldBindJSON** akan mereturn sebuah **error** jika memang terjadi sebuah **error**, maka dari itu kita perlu memvalidasi terlebih dahulu jika terjadi sebuah error.

Create Car

Kemudian method **AbortWithError** pada baris **23** digunakan untuk melempar error jika memang ada error yang terjadi. Method **AbortWithError** menerima 2 parameter. Parameter pertama adalah status error nya, kemudian parameter kedua adalah data error nya.

```
19  func CreateCar(ctx *gin.Context) {
20    var newCar Car
21
22    if err := ctx.ShouldBindJSON(&newCar); err != nil {
23      ctx.AbortWithError(http.StatusBadRequest, err)
24      return
25    }
26    newCar.CarID = fmt.Sprintf("c%d", len(CarDatas)+1)
27    CarDatas = append(CarDatas, newCar)
28
29    ctx.JSON(http.StatusCreated, gin.H{
30      "car": newCar,
31    })
32  }
```

Create Car

Kemudian jika tidak terjadi error ketika proses data binding, maka data request body yang telah ditampung oleh variable **newCar**, akan kita tambahkan pada variable global CarDatas.

Pada baris ke 26, kita mencoba untuk mengenarate id untuk tiap data mobil baru.

Setelah selesai ditampung, maka kita mengirim data response kepada client dengan menggunakan method **JSON**. Method **JSON** digunakan untuk mengirim response kepada client dengan format data JSON. Method ini menerima 2 parameter. Parameter pertama adalah status response nya, dan parameter kedua adalah data response yang di kirimkan kepada client.

```
19  func CreateCar(ctx *gin.Context) {
20    var newCar Car
21
22    if err := ctx.ShouldBindJSON(&newCar); err != nil {
23      ctx.AbortWithError(http.StatusBadRequest, err)
24      return
25    }
26    newCar.CarID = fmt.Sprintf("c%d", len(CarDatas)+1)
27    CarDatas = append(CarDatas, newCar)
28
29    ctx.JSON(http.StatusCreated, gin.H{
30      "car": newCar,
31    })
32  }
```

Router

Setelah membuat endpoint untuk membuat data mobil baru, maka kita perlu membuat routing yang akan menghubungkan client kepada endpoint tersebut.

Buatlah sebuah file dengan nama **carRoute.go** pada folder **routers**. Kemudian pada file tersebut, isilah dengan kode-kode seperti pada gambar kedua dibawah.


```go
package routers

import (
  "belajar-gin/controllers"

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

func StartServer() *gin.Engine {
  router := gin.Default()

  router.POST("/cars", controllers.CreateCar)

  return router
}
```

Router

Function **StartServer** akan kita gunakan untuk menjalankan server dari aplikasi kita. Function ini mengembalikan suatu data dengan tipe data struct ***gin.Engine** yang berasal dari **Gin**, dan kita gunakan untuk menjalankan server, sebagai multiplexer dari routing dan lain-lain.

Pada baris ke 10, variable **router** kita gunakan sebagai penampung untuk engine dari router yang kita dapatkan dari pemanggilan function **gin.Default**.

```
1   package routers
2
3   import (
4     "belajar-gin/controllers"
5
6     "github.com/gin-gonic/gin"
7   )
8
9   func StartServer() *gin.Engine {
10    router := gin.Default()
11
12    router.POST("/cars", controllers.CreateCar)
13
14    return router
15  }
```

Router

Pada baris ke 12, kita menggunakan method **POST** untuk menghubungkan client dengan endpoint **CreateCar**. Kita memberikan 2 parameter terhadapt method **POST** ini. Parameter pertama adalah route path nya, sedangkan parameter kedua memerlukan handler atau endpoint nya. Endpoint harus merupakan sebuah function yang menerima satu parameter dengan tipe data ***gin.Context**.

Karena kita sudah membuat function bernama **CreateCar** yang menerima satu parameter yang sesuai dengan kriteria method **POST**, maka kita dapat menjadi function **CreateCar** untuk parameter kedua dari method **POST**.

```
1   package routers
2
3   import (
4     "belajar-gin/controllers"
5
6     "github.com/gin-gonic/gin"
7   )
8
9   func StartServer() *gin.Engine {
10    router := gin.Default()
11
12    router.POST("/cars", controllers.CreateCar)
13
14    return router
15  }
```

Update Car

Sekarang kita akan membuat proses untuk mengupdate data mobil.

Buatlah sebuah function dengan nama **UpdateCar** pada file **carController.go** seperti pada gambar di sebelah kanan.

Pada baris **35**, method **ctx.Param** merupakan sebuah method yang digunakan untuk mendapat **request parameter** yang dikirimkan oleh client.

Method **ctx.Param** menerima satu parameter dimana kita perlu meletekkan nama parameter yang kita buat pada router nya nanti.

```
34  func UpdateCar(ctx *gin.Context) {
35    carID := ctx.Param("carID")
36    condition := false
37    var updatedCar Car
38
39    if err := ctx.ShouldBindJSON(&updatedCar); err != nil {
40      ctx.AbortWithError(http.StatusBadRequest, err)
41      return
42    }
43
44    for i, car := range CarDatas {
45      if carID == car.CarID {
46        condition = true
47        CarDatas[i] = updatedCar
48        CarDatas[i].CarID = carID
49        break
50      }
51    }
52
53    if !condition {
54      ctx.AbortWithStatusJSON(http.StatusNotFound, gin.H{
55        "error_status":  "Data Not Found",
56        "error_message": fmt.Sprintf("car with id %v not found", carID),
57      })
58      return
59    }
60
61    ctx.JSON(http.StatusOK, gin.H{
62      "message": fmt.Sprintf("car with id %v has been successfully updated", carID),
63    })
64  }
```

Update Car

Kemudian sebelum kita betul-betul mengupdate data mobilnya, pada baris **44** - **50**, kita melooping data-data mobil yang ditampung oleh variable **CarDatas** guna untuk mencari terlebih dahulu data mobil yang ingin di update oleh client nantinya.

Kita mencari data mobilnya berdasarkan id data mobil yang dikirmkan client melalui **request parameter**. Jika data mobilnya tidak dapat, maka kita langsung melempar error dengan status **Data Not Found(404)**.

Namun jika data mobilnya ada, maka kita langsung mengupdate data mobilnya dan menghentikan proses loopingnya.

```
34  func UpdateCar(ctx *gin.Context) {
35    carID := ctx.Param("carID")
36    condition := false
37    var updatedCar Car
38
39    if err := ctx.ShouldBindJSON(&updatedCar); err != nil {
40      ctx.AbortWithError(http.StatusBadRequest, err)
41      return
42    }
43
44    for i, car := range CarDatas {
45      if carID == car.CarID {
46        condition = true
47        CarDatas[i] = updatedCar
48        break
49      }
50    }
51
52    if !condition {
53      ctx.AbortWithStatusJSON(http.StatusNotFound, gin.H{
54        "error_status":  "Data Not Found",
55        "error_message": fmt.Sprintf("car with id %v not found", carID),
56      })
57      return
58    }
59
60    ctx.JSON(http.StatusOK, gin.H{
61      "message": fmt.Sprintf("car with id %v has been successfully updated", carID),
62    })
63  }
```

HACKTIV8

Update Car

Sekarang kita perlu meregistrasikan endpoint **UpdateCar** pada function **StartServer** yang terletak pada file
carRouter.go di dalam folder **routers**.

```
9   func StartServer() *gin.Engine {
10    router := gin.Default()
11
12    router.POST("/cars", controllers.CreateCar)
13
14    router.PUT("/cars/:carID", controllers.UpdateCar)
15
16    return router
17  }
```

Get Car

Buatlah sebuah function bernama **GetCar** yang akan kita gunakan untuk endpoint mendapatkan satu data mobil.

Buatlah function **GetCar** tersebut dengan kode-kode seperti pada gambar di sebelah kanan.

```
66  func GetCar(ctx *gin.Context) {
67    carID := ctx.Param("carID")
68    condition := false
69    var carData Car
70
71    for i, car := range CarDatas {
72      if carID == car.CarID {
73        condition = true
74        carData = CarDatas[i]
75        break
76      }
77    }
78
79    if !condition {
80      ctx.AbortWithStatusJSON(http.StatusNotFound, gin.H{
81        "error_status":  "Data Not Found",
82        "error_message": fmt.Sprintf("car with id %v not found", carID),
83      })
84      return
85    }
86
87    ctx.JSON(http.StatusOK, gin.H{
88      "car": carData,
89    })
90  }
```

HACKTIV8

Get Car

Kemudian kita perlu meregistrasikan endpoint **GetCar** pada function StartServer seperti endpoint-endpoint sebelumnya.

```
 9  func StartServer() *gin.Engine {
10    router := gin.Default()
11
12    router.POST("/cars", controllers.CreateCar)
13
14    router.PUT("/cars/:carID", controllers.UpdateCar)
15
16    router.GET("/cars/:carID", controllers.GetCar)
17
18    return router
19  }
```

Delete Car

Dan sekarang, kita akan membuat endpoint terakhir kita yaitu endpoint untuk menghapus satu data mobil.

Buatlah sebuah function dengan nama **DeleteCar**
seperti pada gambar di sebelah kanan.

```go
func DeleteCar(ctx *gin.Context) {
  carID := ctx.Param("carID")
  condition := false
  var carIndex int

  for i, car := range CarDatas {
    if carID == car.CarID {
      condition = true
      carIndex = i
      break
    }
  }

  if !condition {
    ctx.AbortWithStatusJSON(http.StatusNotFound, gin.H{
      "error_status":  "Data Not Found",
      "error_message": fmt.Sprintf("car with id %v not found", carID),
    })
    return
  }

  copy(CarDatas[carIndex:], CarDatas[carIndex+1:])
  CarDatas[len(CarDatas)-1] = Car{}
  CarDatas = CarDatas[:len(CarDatas)-1]

  ctx.JSON(http.StatusOK, gin.H{
    "message": fmt.Sprintf("car with id %v has been successfully deleted", carID),
  })
}
```

Delete Car

Kemudian function **DeleteCar** perlu kita registrasikan pada function **StartServer**.

```
9   func StartServer() *gin.Engine {
10    router := gin.Default()
11
12    router.POST("/cars", controllers.CreateCar)
13
14    router.PUT("/cars/:carID", controllers.UpdateCar)
15
16    router.GET("/cars/:carID", controllers.GetCar)
17
18    router.DELETE("/cars/:carID", controllers.DeleteCar)
19
20    return router
21  }
```

HACKTIV8

Func main()

Pada file **main.go**, ikutilah kode diatas untuk menjalankan server nya.

```
1  package main
2
3  import "belajar-gin/routers"
4
5  func main() {
6    var PORT = ":8080"
7
8    routers.StartServer().Run(PORT)
9  }
```

HACKTIV8

Test Endpoint With Postman

Mari kita coba mengirim request melalui client dengan menggunakan Postman. Jalankan server dengan perintah **go run main.go**.

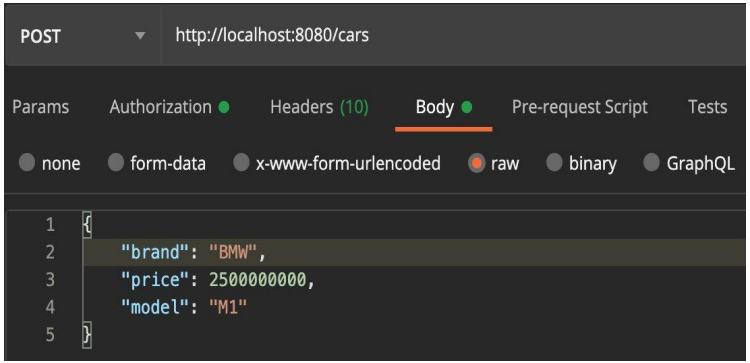
```
└─ go run main.go
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.

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

[GIN-debug] POST   /cars                     --> belajar-gin/controllers.CreateCar (3 handlers)
[GIN-debug] PUT    /cars/:carID              --> belajar-gin/controllers.UpdateCar (3 handlers)
[GIN-debug] GET    /cars/:carID              --> belajar-gin/controllers.GetCar (3 handlers)
[GIN-debug] DELETE /cars/:carID              --> belajar-gin/controllers.DeleteCar (3 handlers)
[GIN-debug] Listening and serving HTTP on :8080
```

Test Endpoint With Postman

Kita akan melakukan test endpoint untuk **Create Car,** kirimkan request seperti pada gambar disamping ini untuk proses membuat data mobil baru.



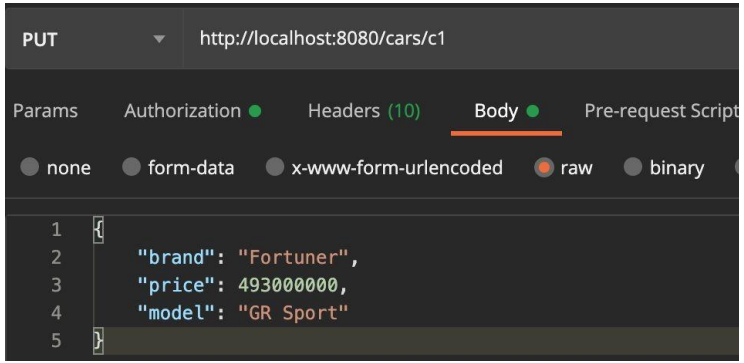

```
{
    "car": {
        "car_id": "c1",
        "brand": "BMW",
        "model": "M1",
        "price": 2500000000
    }
}
```

Dan response yang di dapatkan oleh client akan seperti pada gambar disebelah kiri ini.

HACKTIV8

Test Endpoint With Postman

Kita akan melakukan test endpoint untuk **Update Car,** kirimkan request seperti pada gambar disamping ini untuk proses membuat data mobil baru.



Dan response yang di dapatkan oleh client akan seperti pada gambar dibawah ini.

Test Endpoint With Postman

Mari kita coba lihat datanya dengan melakukan test endpoint **Get Car.**


```
{
    "car": {
        "car_id": "c1",
        "brand": "Fortuner",
        "model": "GR Sport",
        "price": 493000000
    }
}
```

Test Endpoint With Postman

Mari kita coba delete datanya dengan melakukan test endpoint **Delete Car.**


```
{
    "message": "car with id c1 has been successfully deleted"
}
```