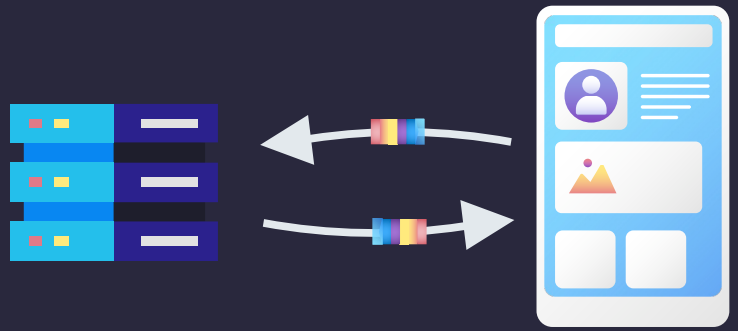
API Design

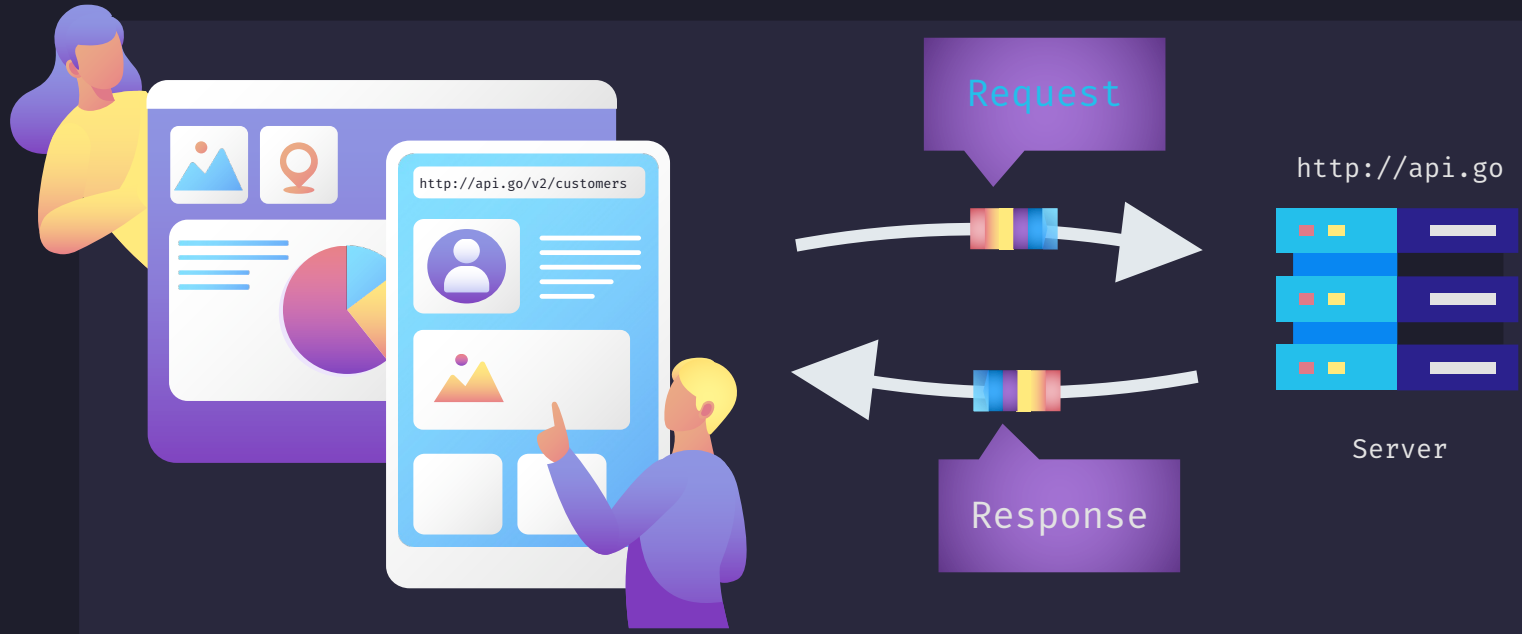


Expose data and application functionality.

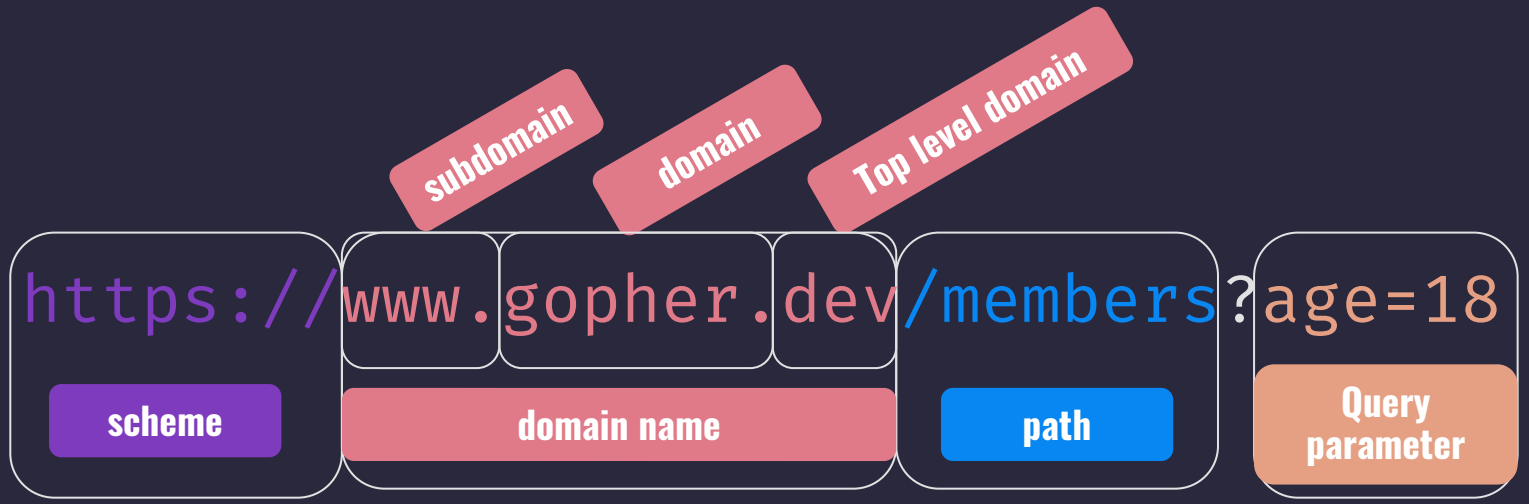
RESTful API

exchange information securely over the http protocol.

REpresentational State Transfer

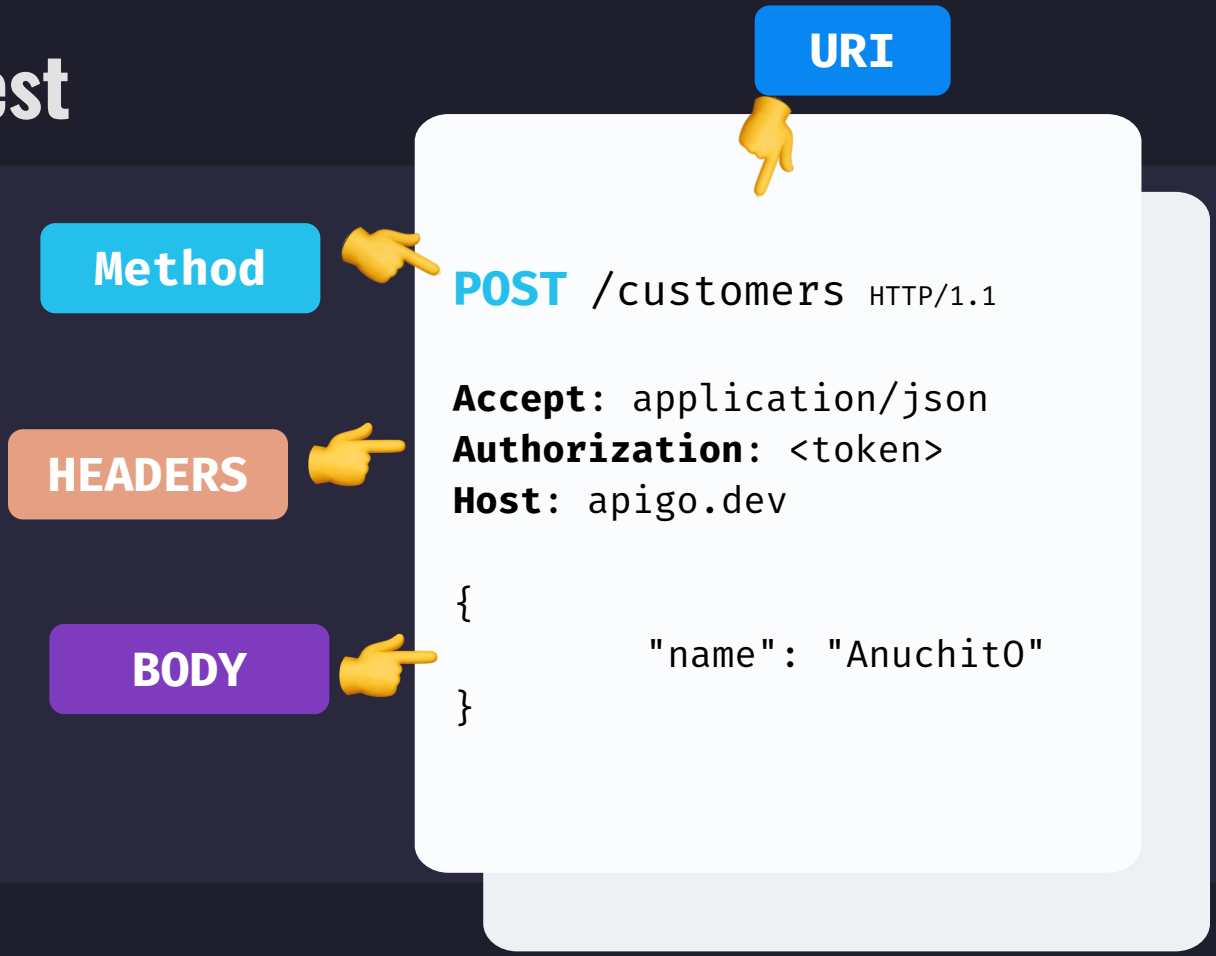


Components of a URL

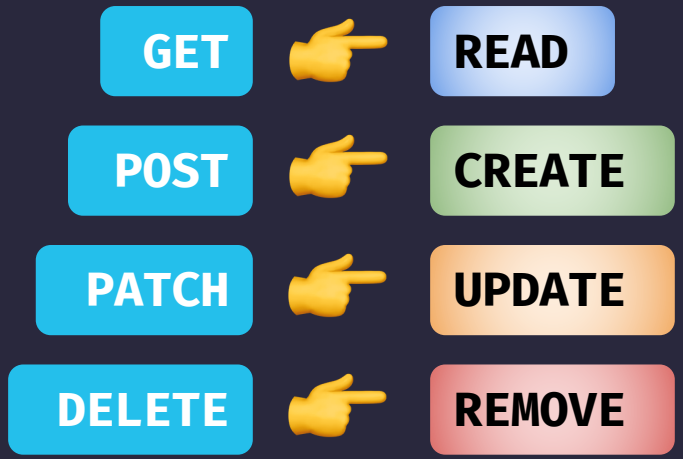


Path & Query parameter

Request


URI
Method
POST /customers HTTP/1.1
HEADERS
Accept: application/json
Authorization: <token>
Host: apigo.dev
BODY
{
"name": "AnuchitO"
}

HTTP request methods (HTTP verbs)



HTTP request methods

`GET`

The `GET` method requests a representation of the specified resource. Requests using `GET` should only retrieve data.

`HEAD`

The `HEAD` method asks for a response identical to a `GET` request, but without the response body.

`POST`

The `POST` method submits an entity to the specified resource, often causing a change in state or side effects on the server.

`PUT`

The `PUT` method replaces all current representations of the target resource with the request payload.

`DELETE`

The `DELETE` method deletes the specified resource.

`CONNECT`

The `CONNECT` method establishes a tunnel to the server identified by the target resource.

`OPTIONS`

The `OPTIONS` method describes the communication options for the target resource.

`TRACE`

The `TRACE` method performs a message loop-back test along the path to the target resource.

`PATCH`

The `PATCH` method applies partial modifications to a resource.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Response

STATUS

HTTP/1.1 200 OK

Server: nginx
Set-Cookie: expires=1136189045
Content-Type: application/json

```
{
        "id": "de0ce2d…",
    "status": "success"
}
```

HEADERS

BODY

HTTP Status

100 to 199	Informational
200 to 299	Successful
300 to 399	Redirection
400 to 499	Invalid Request
500 to 599	SERVER BROKEN

Resources

/customers

/accounts

/funds

http://api.go

Server

RESTful Resource Naming Conventions

Pluralized URIs

Pluralize all resources unless they are singleton resources.

URIs as nouns

REST APIs meant to manipulate resources that should be noun

Forward slashes for hierarchy

show the hierarchy between individual resources and collections.

Query parameters

where necessary such as sort or filter a collection

Do not use file extensions

If you need to specify the format of the body, instead use the Content-Type header.

Lowercase letters and dashes

Resource should use exclusively lowercase letters and dashes (-)

Should

```
/users

/users/{id}

/users/{id}/orders

/users?sort=age

/users
Content-type: application/xml

/users/{id}/pending-orders
```

Should NOT

```
/user

/getUser

/users/{id}\orders

/users/sort/age

/users.xml

/users/{id}/Pending_Orders
```

User resource: get all users

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Request

```
GET /users
```

Response

```
[
  {"id": "u1", "name": "AnuchitO", "age": 18},
  {"id": "u2", "name": "Aorjoa", "age": 18},
  {"id": "u3", "name": "Nong", "age": 26}
]
```

User resource: filter by age

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Request

```
GET /users?age=18
```

Response

```
[
  {"id": "u1", "name": "AnuchitO", "age": 18},
  {"id": "u2", "name": "Aorjoa", "age": 18}
]
```

User resource: get by user ID : /users/{id}

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Request

```
GET /users/u2
```

Response

```
{"id": "u2", "name": "Aorjoa", "age": 18}
```

Example: Order resource

Request

```
GET /orders
```

Response

```
[
  {"id": "o1", "item": "iPhone", "status": "pending"},
  {"id": "o2", "item": "Samsung", "status": "done"},
  {"id": "o3", "item": "Nokia", "status": "done"}
]
```

Order

ID	item	status
o1	iPhone	pending
o2	Samsung	done
o3	Nokia	done

Example: Order resource

Request

```
GET /orders?status=done
```

Response

```
[

  {"id": "o2", "item": "Samsung", "status": "done"},
  {"id": "o3", "item": "Nokia", "status": "done"}
]
```

Order

ID	item	status
o1	iPhone	pending
o2	Samsung	done
o3	Nokia	done

Order resource: get by user ID : /orders/{id}

Request

```
GET /orders/o3
```

Response

```
{"id": "o3", "item": "Nokia", "status": "done"}
```

Order

ID	item	status
o1	iPhone	pending
o2	Samsung	done
o3	Nokia	done

User order mobile phone

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Order

ID	item	status	UserID
o1	iPhone	pending	u1
o2	Samsung	done	u1
o3	Nokia	done	u2

> Which products does **AnuchitO** has been order?

> Which products does **AnuchitO** has been ordered?

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Order

ID	item	status	UserID
o1	iPhone	pending	u1
o2	Samsung	done	u1
o3	Nokia	done	u2

```
GET /orders?userID=u1

[
  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"},
  {"id": "o2", "item": "Samsung", "status": "done", userID: "u1"}
]

GET /users/u1/orders

[
  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"},
  {"id": "o2", "item": "Samsung", "status": "done", userID: "u1"}
]
```

> Which **AnuchitO**’s orders are pending?

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Order

ID	item	status	UserID
o1	iPhone	pending	u1
o2	Samsung	done	u1
o3	Nokia	done	u2

GET /orders?**userID=u1&status=pending**


```
[
  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"}
]
```


GET /users/u1/orders?**status=pending**


```
[
  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"}
]
```

> what is that difference?

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Order

ID	item	status	UserID
o1	iPhone	pending	u1
o2	Samsung	done	u1
o3	Nokia	done	u2

```
GET /orders/o1

  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"}


GET /users/u1/orders/o1

  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"}
```

> what is that difference?

User

ID	name	age
u1	AnuchitO	18
u2	Aorjoa	18
u3	Nong	26

Order

ID	item	status	UserID
o1	iPhone	pending	u1
o2	Samsung	done	u1
o3	Nokia	done	u2

GET /orders/o1

```
  {"id": "o1", "item": "iPhone", "status": "pending", userID:
"u1"}
```

GET /users/**u2**/orders/o1

No data

Exercise

Let’s design **api path** for Customer resources base on four operations CREATE, UPDATE, READ, DELETE.

Exercise

Let’s design **api path** for Customer resources base on four operations CREATE, UPDATE, READ, DELETE.

GET /customers	get all customers
GET /customers/{id}	get customer by ID (only one)
POST /customers	create new customer
PATCH /customers/{id}	update some data of customer
(only one)	
DELETE /customers/{id}	datele customer by ID (only one)

net/http

Go’s Standard library provides HTTP functionalities

There are client, server and others

Client

```
tr := &http.Transport{
        MaxIdleConns:      10,
        IdleConnTimeout:   30 *
time.Second,
        DisableCompression:true,
}

client := &http.Client{ Transport:tr}
```

Server

```
s := &http.Server{
        Addr:           ":2565",
        Handler:        myHandler,
        ReadTimeout:    10 *
time.Second,
        WriteTimeout:   10 *
time.Second,
        MaxHeaderBytes: 1 << 20,
}

log.Fatal(s.ListenAndServe())
```

Simple http server

```
func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
                w.Write([]byte(`{"name": "anuchito"}`))
        })

        log.Fatal(http.ListenAndServe(":2565", nil))
}
```

Handle http method

```
func main() {
	http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
		if r.Method == "GET" {
			w.Write([]byte(`{"name": "anuchito", "method":
"GET"}`))
			return
		}

		w.WriteHeader(http.StatusMethodNotAllowed)
	})

	log.Fatal(http.ListenAndServe(":2565", nil))
}
```

JSON in GO

Package json implements encoding and decoding of JSON

Struct to JSON

```
import "encoding/json"

type User struct {
            ID   int    `json:"id"`
            Name string `json:"name"`
            Age  int    `json:"age"`
}

func main() {
            t := User{ ID: 1, Name: "AnuchitO", Age: 18 }

            b, err := json.Marshal(t)

            fmt.Printf("type : %T \n", b)                  // type : []uint8
            fmt.Printf("byte : %v  \n", b)                 // byte : [123 34 105 100 …]
            fmt.Printf("string: %s \n", b)                 // string:
{"id":1,"name":"AnuchitO","age":18}
            fmt.Println(err)                                          // <nil>
}
```

Response all User

```
var users = []User{
            {ID: 1, Name: "AnuchitO", Age: 18},
}

func usersHandler(w http.ResponseWriter, req *http.Request) {
            if req.Method == "GET" {
                        log.Println("GET")
                        b, err := json.Marshal(users)
                        if err != nil {
                                    w.WriteHeader(http.StatusInternalServerError)
                                    fmt.Fprintf(w, "error: %v", err)
                                    return
                        }

                        w.Header().Set("Content-Type", "application/json")
                        w.Write(b)
            }
}
```

JSON to Struct

```
import "encoding/json"

type User struct {
            ID   int    `json:"id"`
            Name string `json:"name"`
            Age  int    `json:"age"`
}

func main() {
            data := []byte(`{ "id": 2, "name": "Aorjoa", "age": 19 }`)

            var u User
            err := json.Unmarshal(data, &u)

            fmt.Printf("% #v\n", u)            // main.User{ID: 2, Name:"Aorjoa", Age: 19}
            fmt.Println(err)                              // <nil>
}
```

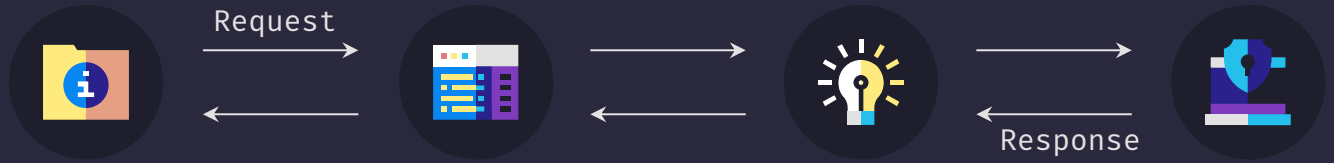
Add more user

```
func usersHandler(w http.ResponseWriter, req *http.Request) {
            if req.Method == "POST" {
                        body, err := ioutil.ReadAll(req.Body)
                        if err != nil {
                                    fmt.Fprintf(w, "error : %v", err)
                                    return
                        }

                        t := User{}
                        err = json.Unmarshal(body, &t)
                        if err != nil {
                                    fmt.Fprintf(w, "error: %v", err)
                                    return
                        }

                        users = append(users, t)
                        fmt.Fprintf(w, "hello %s created users", "POST")
                        return
            }
```

Middleware

Pre and/or post processing of the request

Middleware



Client Go Http Server Middleware Handle Function

First Class Functions

```
type Math func(int, int) int

func cal(sn Math) int {
        return sn(5, 4)
}

func sum(a int, b int) int {
        return a + b
}
```

```
func main() {
        fn := sum
        r1 := fn(1, 2)
        fmt.Println("fn(1,2):", r1) // fn(1,2): 3

        r2 := cal(fn)
        fmt.Println("cal(fn):", r2) // cal(fn): 9

        r3 := cal(sum)
        fmt.Println("cal(sum):", r3) // cal(sum):
9
}
```

Function literals

```
import "fmt"

func main() {
        r := func(a, b int) bool
{
                return a < b
        }(2, 3)

        fmt.Println("result:",
r)
}
```

```
func main() {
        go func(a, b int) {
                dosomthing(a, b)
        }(3, 4)
}
```

Higher-order functions

```
type Decorator func(s string) error

func Use(next Decorator) Decorator {
        return func(c string) error {
                fmt.Println("do something
before")
                r := c + " should be green."
                return next(r)
        }
}

func home(s string) error {
        fmt.Println("home", s)
        return nil
}
```

```
func main() {
        wrapped := Use(home)
        w := wrapped("world")
        fmt.Println("end result:",
w)
}
```

Middleware Individual endpoint

```
func log(next http.HandlerFunc) http.HandlerFunc {
          return http.HandlerFunc(func(w
http.Res...) {
                    start := time.Now()
                    next.ServeHTTP(w, r)
                    log.Printf("Server http
middleware...")
          })
}
```

```
// function main
http.HandleFunc("/users", log(usersHandler))

http.HandleFunc("/health", log(healthHandler))
```

Middleware with Mux

```
type Logger struct {
        Handler http.Handler
}

func (l Logger) ServeHTTP(w http.Re...) {
        start := time.Now()
        l.Handler.ServeHTTP(w, r)
        log.Printf("Server http middleware:...)
}
```

```
mux := http.NewServeMux()
        mux.HandleFunc("/users", users
        mux.HandleFunc("/health",
health)

        logMux := Logger{Handler:mux}

        srv := http.Server{
                Addr:    ":2565",
                Handler: logMux,
        }
```

Basic Auth

Example of Authentication

Authorization header

Client Basic Auth

```
GET /users HTTP/1.1
Authorization: Basic
YXBpZGVzaWduOjQ1Njc4
Host: localhost:2565
```

Server Basic Auth

```
func AuthMiddleware(w http.ResponseWriter, r *http.Request) {
            u, p, ok := r.BasicAuth()
            if !ok {
                        w.Write([]byte(`can't parse the basic
auth`))
                        w.WriteHeader(401)
                        return
            }

            if u != "apidesign" || p != "45678" {
                        w.Write([]byte(`Username/Password
incorrect.`))
                        w.WriteHeader(401)
                        return
            }

            fmt.Println("Auth passed.")
            w.WriteHeader(200)
            return
}
```

Auth Middleware

```
func AuthMiddleware(next http.HandlerFunc) http.HandlerFunc {
            return func(w http.ResponseWriter, req *http.Request) {
                        u, p, ok := req.BasicAuth()
                        if !ok {
                                    w.WriteHeader(401)
                                    w.Write([]byte(`can't parse the basic auth`))
                                    return
                        }

                        if u != "apidesign" || p != "45678" {
                                    w.WriteHeader(401)
                                    w.Write([]byte(`Username/Password incorrect.`))
                                    return
                        }

                        fmt.Println("Auth passed.")
                        next(w, req)
                        return
            }
}
```

HTTP API framework : Echo

High performance, extensible, minimalist Go web framework

Echo framework

```
func main() {
        e := echo.New()

        e.Use(middleware.Logger())
        e.Use(middleware.Recover())

        e.GET("/users", func(c echo.Context) error {
                return c.JSON(http.StatusOK, users)
    })

        log.Fatal(e.Start(":2565"))
}
```

Group

```
func main() {
	e := echo.New()
	e.Use(middleware.Logger())
	e.Use(middleware.Recover())

	e.GET("/health", func(c echo.Context) error {
		return c.String(http.StatusOK, "OK")
	})

	g := e.Group("/api")
	g.Use(middleware.BasicAuth(AuthMiddleware))
	g.POST("/users", createUserHandler)
	g.GET("/users", getUsersHandler)

	log.Fatal(e.Start(":2565"))
}
```

SQL Database

Connect database

```
import (
 "database/sql"
 "log"
 "os"
 _ "github.com/lib/pq"
)

func main() {
 //db, err := sql.Open("postgres", "root:password@tcp(127.0.0.1:3306)/dbname")
 db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))

 if err != nil { log.Fatal("Connect to database error", err)}
 defer db.Close()

 log.Println("okay")
}
```

Register driver

```
func init() {
        sql.Register("postgres", &Driver{})
}
```

Create table

```
func main() {
            db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))
            if err != nil {
                        log.Fatal("Connect to database error", err)
            }
            defer db.Close()

            createTb := `
            CREATE TABLE IF NOT EXISTS users ( id SERIAL PRIMARY KEY, name TEXT, age INT );
            `

            _, err = db.Exec(createTb)
            if err != nil {
                        log.Fatal("can't create table", err)
            }

            fmt.Println("create table success")
}
```

Insert

```
func main() {
	db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))
	if err != nil {
		log.Fatal("Connect to database error", err)
	}
	defer db.Close()

	row := db.QueryRow("INSERT INTO users (name, age) values ($1, $2)  RETURNING id",
"AnuchitO", 19)
	var id int
	err = row.Scan(&id)
	if err != nil {
		fmt.Println("can't scan id", err)
		return
	}

	fmt.Println("insert todo success id : ", id)
}
```

Query all

```
stmt, err := db.Prepare("SELECT id, name, age FROM users")
	if err != nil {
		log.Fatal("can't prepare query all users statment", err)
	}

	rows, err := stmt.Query()
	if err != nil {
		log.Fatal("can't query all users", err)
	}
	for rows.Next() {
		var id, age int
		var name string
		err := rows.Scan(&id, &name, &age)
		if err != nil {
			log.Fatal("can't Scan row into variable", err)
		}
		fmt.Println(id, name, age)
	}
```

Query one row

```
stmt, err := db.Prepare("SELECT id, name, age FROM users where id=$1")
     if err != nil {
               log.Fatal("can'tprepare query one row statment", err)
     }

     rowId := 1
     row := stmt.QueryRow(rowId)
     var id, age int
     var name string

     err = row.Scan(&id, &name, &age)
     if err != nil {
               log.Fatal("can't Scan row into variables", err)
     }

     fmt.Println("one row", id, name, age)
```

Update

```
stmt, err := db.Prepare("UPDATE users SET name=$2 WHERE id=$1;")

    if err != nil {
            log.Fatal("can't prepare statment update", err)
    }

    if _, err := stmt.Exec(1, "Aorjoa"); err != nil {
            log.Fatal("error execute update ", err)
    }

    fmt.Println("update success")
```

Delete

```
stmt, err := db.Prepare("DELETE FROM users WHERE id = $1")
     if err != nil {
               log.Fatal("can't prepare delete statement", err)
     }

     if _, err := stmt.Exec(1); err != nil {
               log.Fatal("can't execute delete statment", err)
     }
```

Exercise

- Store Users in database table name `users` when call POST /users with body eg.

```
{
  "name": "Aorjoa",
  "age": 19
}
```

- Get all users from database table `users` when call GET /users
- Get user by ID when call api with path params GET /users/:id

API Integration test

Http client

```
req, _ := http.NewRequest(method, url, body)
req.Header.Add("Authorization", "November 10, 2009")
req.Header.Add("Content-Type", "application/json")
client := http.Client{}
res, err := client.Do(req)
```

Build tag

```
//go:build integration

package main

import (
...
```

Graceful shutdown

Goroutine

```
func slow(s string) {
        for i := 0; i < 3; i++ {
                time.Sleep(1 *
time.Second)
                fmt.Println(s,
":", i)
        }
}
```

```
func main() {
        go slow("gopher")

        slow("nong")

        time.Sleep(10 * time.Second)
        fmt.Println("all task done.")
}
```

Channels

```
func slow(s string) {
        for i := 0; i < 3; i++ {
                time.Sleep(1 *
time.Second)
                fmt.Println(s,
":", i)
        }
}
```

```
func main() {
        done := make(chan bool)

        go func() {
                slow("gopher")
                done <- true
        }()

        slow("nong")

        <-done
        fmt.Println("all task done.")
}
```

signal.Notify

```
package main

import (
        "log"
        "os"
        "os/signal"
        "syscall"
)

func main() {
        log.Println("Server started")
        stop := make(chan os.Signal, 1)
        signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
        log.Println("wait for signal")
        <-stop
        log.Println("Server Stopped")
}
```

Graceful shutdown: net/http

```
func main() {
            mux := http.NewServeMux()
            srv := http.Server{
                        Addr:    ":2565",
                        Handler: mux,
            }

            go func() { log.Fatal(srv.ListenAndServe()) }()
            fmt.Println("server starting at :2565")
            shutdown := make(chan os.Signal, 1)
            signal.Notify(shutdown, os.Interrupt, syscall.SIGTERM)
            <-shutdown
            fmt.Println("shutting down...")
            if err := srv.Shutdown(context.Background()); err != nil {
                        fmt.Println("shutdown err:", err)
            }
            fmt.Println("bye bye")
}
```

Graceful shutdown: Echo

```
func main() {
            e := echo.New()
            e.GET("/", func(c echo.Context) error { return c.JSON(http.StatusOK, "OK") })
            go func() {
                        if err := e.Start(":1323"); err != nil && err != http.ErrServerClosed { //
Start server
                                    e.Logger.Fatal("shutting down the server")
                        }
            }()

            shutdown := make(chan os.Signal, 1)
            signal.Notify(shutdown, os.Interrupt, syscall.SIGTERM)
            <-shutdown
      fmt.Println("shutting down...")
            ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
            defer cancel()
            if err := e.Shutdown(ctx); err != nil {
                        e.Logger.Fatal(err)
            }
}
```

Go Packages

Exported

In Go, a name is exported if it begins with a **capital letter**. For example, `**H**ello` is an exported name but `**h**ello` will be private member under that package can only see.

How to build Go

Go build tools

```
$ go build
```
generate an executable for the current platform and architecture

```
$ go tool dist list
```
To find this list of possible platforms

```
$ GOOS=linux GOARCH=amd64 go build
```
build with specific OS, ARCH

Build go with Docker

```
// Dockerfile
FROM golang:1.19-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY *.go ./

RUN go build -o /app

EXPOSE 2565

CMD [ "/app" ]
```

```
// BUILD
$ docker build -t go-app .


// RUN
$ docker run go-app
```