Go

ODDS | Thaibev

Day 2

Coverage

- Go Advanced Concepts
- What is an API?
- Echo, minimalist Go Web Framework
- Configuration Management with Viper

Go Advanced Concepts

Go Advanced Concepts

- First Class Function
- Anonymous Function
- Higher-Order Function
- Function Closures

First Class Function

```
func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(5, 12))
}
```

Anonymous Function

An anonymous function is a function which doesn’t contain any name. It is useful when you want to create an inline function. In Go language, an anonymous function can form a closure

Anonymous Function

```go
package main

import "fmt"

func main() {
  // Anonymous function
  func(){
      fmt.Println("vim-go")
  }()
}
```

Higher-Order Function

- Take one or more functions as arguments

```
compute(math.Pow)

func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}
```

Higher-Order Function

- Returns a function as its result

```
func giveMeAFunc() func(string) {
    return func(message string){
        fmt.Println(message)
    }
}
```

Function closures

```
func incrementor() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

Function closures

```
func main() {
    next := incrementor()  // next is a
function returned by incrementor
    fmt.Println(next())    // prints 1
    fmt.Println(next())    // prints 2
    fmt.Println(next())    // prints 3
}
```

Exercise

the `adder` function is a higher-order function that takes a function f as an argument. The function f takes an int as an argument and returns an int as a result. The `adder` function then calls the f function with the argument 2, and returns the result. In the main function, please define a function `add10` which takes an int and returns the int incremented by 10, and passed it as argument to `adder` function, which then call the function `add10` with argument 2 and print the result 12.

Exercise

the filter function is a higher-order function that takes a slice of int and a function f as arguments. The function f takes an int as an argument and returns a bool as a result. The filter function then iterates over the slice of data, and for each item, it calls the f function with the current item as an argument. If the f function returns true, the current item is added to the result slice. In the main function, we define a function even which takes an int and returns true if the number is even and false otherwise, and passed it as argument to filter function, which then filters the slice of data and return the even numbers.

What is an API?

What is an API?

An application programming interface is a way for two or more computer programs to communicate with each other.

Why should be separate front & back-end?

- Scalability
- Resource optimization
- Easier upgradation
- Simpler to switch frameworks
- Faster Deployment
- Consolidation of APIs
- Modularity

Echo, minimalist Go Web Framework

https://echo.labstack.com/guide/

Installation

```
$ mkdir myapp && cd myapp
$ go mod init myapp
$ go get github.com/labstack/echo/v4
```

https://echo.labstack.com/guide/

Hello, World!

```
package main

import (
    "net/http"

    "github.com/labstack/echo/v4"
)
```

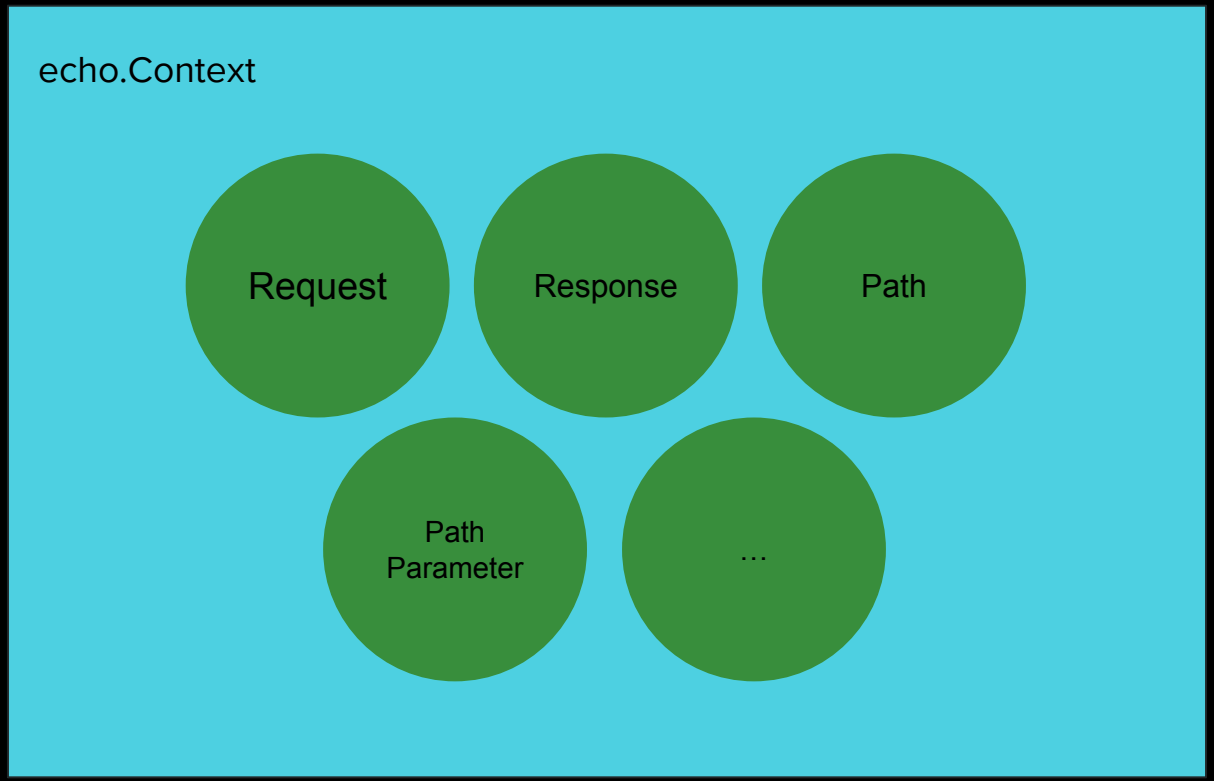
https://echo.labstack.com/guide/

Hello, World!

```
func main() {
    e := echo.New()
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Hello, World!")
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/guide/

Context



Context

`echo.Context` represents the context of the current HTTP request. It holds request and response reference, path, path parameters, data, registered handler and APIs to read request and write response. As Context is an interface.

https://echo.labstack.com/guide/context/

Routing

```
Echo.{HTTP method}(path string, h Handler)

Echo.Any(path string, h Handler)

Echo.Match(methods []string, path string, h
Handler)
```

https://echo.labstack.com/guide/routing/

Routing

```
Echo.{HTTP method}(path string, h Handler)
```

Echo defined handler function as

`func(echo.Context) error`

https://echo.labstack.com/guide/routing/

Routing

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

    // Route: HTTP method, path, handler
    e.GET("/hello", func(c echo.Context) error {
        return c.String(http.StatusOK, "Hello, World!")
    })

    e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/guide/routing/

Routing

```
func main() {
    e := echo.New()
    e.GET("/hello", hello)

    e.Logger.Fatal(e.Start(":1323"))
}

func hello(c echo.Context) error {
      return c.String(http.StatusOK, "Hello, World!")
}
```

https://echo.labstack.com/guide/routing/

Match-any

Matches zero or more characters in the path.

https://echo.labstack.com/guide/routing/

Match-any

For example, pattern `/coupons/*` will match:

- `/coupons/`
- `/coupons/1`
- `/coupons/1/files/1`
- `/coupons/anything...`

https://echo.labstack.com/guide/routing/

Path Matching Order

- Static
- Param
- Match any

https://echo.labstack.com/guide/routing/

Path Matching Order

```
e.GET("/coupons/:id", func(c echo.Context) error {
    return c.String(http.StatusOK, "/coupons/:id")
})

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

e.GET("/coupons/1/files/*", func(c echo.Context) error {
    return c.String(http.StatusOK, "/coupons/1/files/*")
})
```

https://echo.labstack.com/guide/routing/

Group

Routes with common prefix can be grouped to define a new sub-router with optional middleware.

```
Echo#Group(prefix string, m ...Middleware) *Group
```

```
g := e.Group("/admin")
```

https://echo.labstack.com/guide/routing/

Group

```
func main() {
    e := echo.New()
    e.GET("/coupons/:id", func(c echo.Context) error {
        return c.String(http.StatusOK, "/coupons/:id")
    })

    g := e.Group("/admin")
    g.GET("/coupons/:id", func(c echo.Context) error {
        return c.String(http.StatusOK, "/admin/coupons/:id")
    })

    e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/guide/routing/

Request

- Form Data
- Query parameter
- Path parameter
- Binding Data

https://echo.labstack.com/guide/request/

Form Data

Form data can be retrieved by name using `Context#FormValue(name string)`.

```
func(c echo.Context) error {
  name := c.FormValue("name")
  return c.String(http.StatusOK, name)
}
```

https://echo.labstack.com/guide/request/

Form Data

```
func upload(c echo.Context) error {
    // Read form fields
    name := c.FormValue("name")
    email := c.FormValue("email")

    file, err := c.FormFile("file")
    if err != nil {
        return err
    }
    // Do something…
    return c.String(http.StatusOK, fmt.Sprintf("File %s uploaded successfully with
fields name=%s and email=%s.", file.Filename, name, email))
}
```

https://echo.labstack.com/guide/request/

Form Data

```
func upload(c echo.Context) error {
    // Multipart form
    form, err := c.MultipartForm()
    if err != nil {
        return err
    }
    files := form.File["files"]

    for _, file := range files {
     // Do something…
    }
    return c.String(http.StatusOK, fmt.Sprintf("Uploaded successfully %d files with
fields name=%s and email=%s", len(files), name, email))
}
```

https://echo.labstack.com/guide/request/

Query parameter

Query parameters can be retrieved by name using Context#QueryParam(name string).

```
func(c echo.Context) error {
  name := c.QueryParam("name")
  return c.String(http.StatusOK, name)
})
```

https://echo.labstack.com/guide/request/

Path parameter

Registered path parameters can be retrieved by name using Context#Param(name string).

```
e.GET("/coupons/:name", func(c echo.Context) error {
  name := c.Param("name")
  return c.String(http.StatusOK, name)
})
```

https://echo.labstack.com/guide/request/

Header

Value in HTTP Header can be retrieved by name using Context#Request().Header.Get(name string).

```
e.GET("/coupons", func(c echo.Context) error {
  userID := c.Request().Header.Get("USER_ID")
  return c.String(http.StatusOK, userID)
})
```

Binding Data

Also binding of request data to native Go structs and variables is supported

https://echo.labstack.com/guide/request/

Binding

- URL Path parameter
- URL Query parameter
- Header
- Request body

https://echo.labstack.com/guide/binding/

Struct Tag Binding

With struct binding you define a Go struct with tags specifying the data source and corresponding key. In your request handler you simply call `Context#Bind(i interface{})` with a pointer to your struct. The tags tell the binder everything it needs to know to load data from the request.

https://echo.labstack.com/guide/binding/

Struct Tag Binding

```
type Coupon struct {
  ID string `query:"id"`
}

func coupons(c echo.Context) error {
    // in the handler for /coupons?id=<couponID>
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Data Sources

- query - query parameter
- param - path parameter (also called route)
- header - header parameter
- json - request body. Uses builtin Go json package for unmarshalling.
- form - form data. Values are taken from query and request body. Uses Go standard library form parsing.

https://echo.labstack.com/guide/binding/

Data Sources : Query

```
type Coupon struct {
  ID string `query:"id"`
}

func coupons(c echo.Context) error {
    // in the handler for /coupons?id=<couponID>
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Data Sources : Param

```
type Coupon struct {
  ID string `param:"id"`
}

func coupons(c echo.Context) error {
    // in the handler for /coupons/:id
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Data Sources : Header

```
type Coupon struct {
  ID string `header:"id"`
}

func coupons(c echo.Context) error {
    // in the handler for /coupons with header “id”: “<couponID>”
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Data Sources : Header

Note that headers is not one of the included sources with `Context#Bind`

https://echo.labstack.com/guide/binding/

Data Sources : JSON

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

func coupons(c echo.Context) error {
    // in the handler for /coupons with body { “id”: “<couponID>” }
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Data Sources : Form

```
type Coupon struct {
  ID string `form:"id"`
}

func coupons(c echo.Context) error {
    // in the handler for /coupons with form-data “id”: “<couponID>”
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Multi Sources

It is possible to specify multiple sources on the same field. In this case request data is bound in this order:

- Path parameters
- Query parameters (only for GET/DELETE methods)
- Request body

https://echo.labstack.com/guide/binding/

Multi Sources

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id"`
}
```

https://echo.labstack.com/guide/binding/

Multi Sources

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id"`
}
func coupons(c echo.Context) error {
    // in the handler for /coupons/<couponID>
    // in the handler for /coupons?id=<couponID>
    // in the handler for /coupons with body { “id”: “<couponID>” }
    var coupon Coupon
    err := c.Bind(&coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Multi Sources

Note that binding at each stage will overwrite data bound in a previous stage. This means if your JSON request contains the query param name=query and body {"name": "body"} then the result will be Coupon{Name: "body"}.

https://echo.labstack.com/guide/binding/

Direct Source

It is also possible to bind data directly from a specific source:

https://echo.labstack.com/guide/binding/

Direct Source : Request body

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id"`
}

func coupons(c echo.Context) error {
    // try to send in the handler for /coupons/<couponID>
    // try to send in the handler for /coupons?id=<couponID>
    // try to send in the handler for /coupons with body { “id”: “<couponID>” }
    var coupon Coupon
    err := err := (&echo.DefaultBinder{}).BindBody(c, &coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Direct Source : Query parameters

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id"`
}

func coupons(c echo.Context) error {
    // try to send in the handler for /coupons/<couponID>
    // try to send in the handler for /coupons?id=<couponID>
    // try to send in the handler for /coupons with body { “id”: “<couponID>” }
    var coupon Coupon
    err := err := (&echo.DefaultBinder{}).BindQueryParams(c, &coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Direct Source : Path parameters

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id"`
}

func coupons(c echo.Context) error {
    // try to send in the handler for /coupons/<couponID>
    // try to send in the handler for /coupons?id=<couponID>
    // try to send in the handler for /coupons with body { “id”: “<couponID>” }
    var coupon Coupon
    err := err := (&echo.DefaultBinder{}).BindPathParams(c, &coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Direct Source : Header parameters

```
type Coupon struct {
  ID string `param:"id" query:"id" form:"id" json:"id" header:”id”`
}

func coupons(c echo.Context) error {
    // try to send in the handler for /coupons/<couponID>
    // try to send in the handler for /coupons?id=<couponID>
    // try to send in the handler for /coupons with body { “id”: “<couponID>” }
    // try to send in the handler for /coupons with header “id”: “<couponID>”
    var coupon Coupon
    err := err := (&echo.DefaultBinder{}).BindHeaders(c, &coupon); if err != nil {
        return c.String(http.StatusBadRequest, "bad request")
    }
    return c.String(http.StatusOK, "Get Coupon: " + coupon.ID)
}
```

https://echo.labstack.com/guide/binding/

Security

It is advisable to have a separate struct for binding and map it explicitly to your business struct.

https://echo.labstack.com/guide/binding/

Security

Consider what will happen if your bound struct has an Exported field `IsAdmin bool` and the request body contains `{IsAdmin: true, Name: "hacker"}`.

https://echo.labstack.com/guide/binding/

Security

```
type User struct {
  Name  string `json:"name" form:"name" query:"name"`
  Email string `json:"email" form:"email" query:"email"`
}

type UserDTO struct {
  Name    string
  Email   string
  IsAdmin bool
}
```

https://echo.labstack.com/guide/binding/

Security

```
e.POST("/users", func(c echo.Context) (err error) {
  var u User
  if err = c.Bind(&u); err != nil {
    return c.String(http.StatusBadRequest, "bad request")
  }

  // Load into separate struct for security
  user := UserDTO{
    Name: u.Name,
    Email: u.Email,
    IsAdmin: false // avoids exposing field that should not be bound
  }

  executeSomeBusinessLogic(user)

  return c.JSON(http.StatusOK, u)
})
```

https://echo.labstack.com/guide/binding/

Fluent Binding

Echo provides an interface to bind explicit data types from a specified source. It uses method chaining, also known as a Fluent Interface.

https://echo.labstack.com/guide/binding/

Fluent Binding

- `echo.QueryParamsBinder(c)` - binds query parameters (source URL)
- `echo.PathParamsBinder(c)` - binds path parameters (source URL)
- `echo.FormFieldBinder(c)` - binds form fields (source URL + body). See also `Request.ParseForm`.

https://echo.labstack.com/guide/binding/

Fluent Binding : Error Handling

```
type Opts struct {
    Active bool
}

func main() {
    e := echo.New()
    // example param /search?amount=10&active=true
    e.GET("/search", func(c echo.Context) error {
        var opts Opts
        amount := int64(50) // default is 50

        . . .

        return c.String(http.StatusOK, fmt.Sprintf("Active: %v, Amount: %d", opts.Active, amount))
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/guide/binding/

Fluent Binding : Error Handling

```
type Opts struct {
    Active bool
}

func main() {
        . . .
        err := echo.QueryParamsBinder(c).
            Int64("amount", &amount).
            Bool("active", &opts.Active).
            BindError()
        if err != nil {
            return c.String(http.StatusBadRequest, err.Error())
        }
        . . .
}
```

https://echo.labstack.com/guide/binding/

Validate Data

Echo doesn’t have built-in data validation capabilities, however, you can register a custom validator using `Echo#Validator` and leverage third-party libraries.

https://echo.labstack.com/guide/request/#validate-data

Validate Data

Example below uses

https://github.com/go-playground/validator

framework for validation:

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
package main

import (
  "net/http"

  "github.com/go-playground/validator"
  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
)
```

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
type (
  Coupon struct {
    Name        string `json:"name" validate:"required"`
    Description string `json:"description"`
  }

  CustomValidator struct {
    validator *validator.Validate
  }
)
```

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
func (cv *CustomValidator) Validate(i interface{}) error {
  if err := cv.validator.Struct(i); err != nil {
    // Optionally, you could return the error to give each route
more control over the status code
    return echo.NewHTTPError(http.StatusBadRequest, err.Error())
  }
  return nil
}
```

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
func main() {
  e := echo.New()
  e.Validator = &CustomValidator{validator:
validator.New()}

  ...

  e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
e.POST("/coupons", func(c echo.Context) (err error) {
  var coupon Coupon
  if err = c.Bind(coupon); err != nil {
    return echo.NewHTTPError(http.StatusBadRequest, err.Error())
  }

  ...

  return c.JSON(http.StatusOK, coupon)
})
```

https://echo.labstack.com/guide/request/#validate-data

Validate Data

```
e.POST("/coupons", func(c echo.Context) (err error) {
  var coupon Coupon
  if err = c.Bind(coupon); err != nil {
    return echo.NewHTTPError(http.StatusBadRequest, err.Error())
  }
  if err = c.Validate(coupon); err != nil {
    return err
  }
  return c.JSON(http.StatusOK, coupon)
})
```

https://echo.labstack.com/guide/request/#validate-data

Response : Send String

`Context#String(code int, s string)` can be used to send plain text response with status code.

```
func(c echo.Context) error {
  return c.String(http.StatusOK, "Hello, World!")
}
```

Response : Send JSON

`Context#JSON(code int, i interface{})` can be used to encode a provided Go type into JSON and send it as response with status code.

Response : Send JSON

```
type Coupon struct {
  Name        string `json:"name"`
  Description string `json:"description"`
}

func(c echo.Context) error {
  coupon := &Coupon{
    Name:        "Save Up to $15",
    Description: "Up to an Additional $15 Off ODDS Online",
  }
  return c.JSON(http.StatusOK, coupon)
}
```

Response : Send No Content

`Context#NoContent(code int)` can be used to send empty body with status code.

```
func(c echo.Context) error {
  return c.NoContent(http.StatusNoContent)
}
```

Middleware

Middleware is a function chained in the HTTP request-response cycle with access to `Echo#Context` which it uses to perform a specific action, for example, logging every request or limiting the number of requests.

Handler is processed in the end after all middleware are finished executing.

Middleware

Middleware registered using Echo#Use() is only executed for paths which are registered after Echo#Use() has been called.

CORS Middleware

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

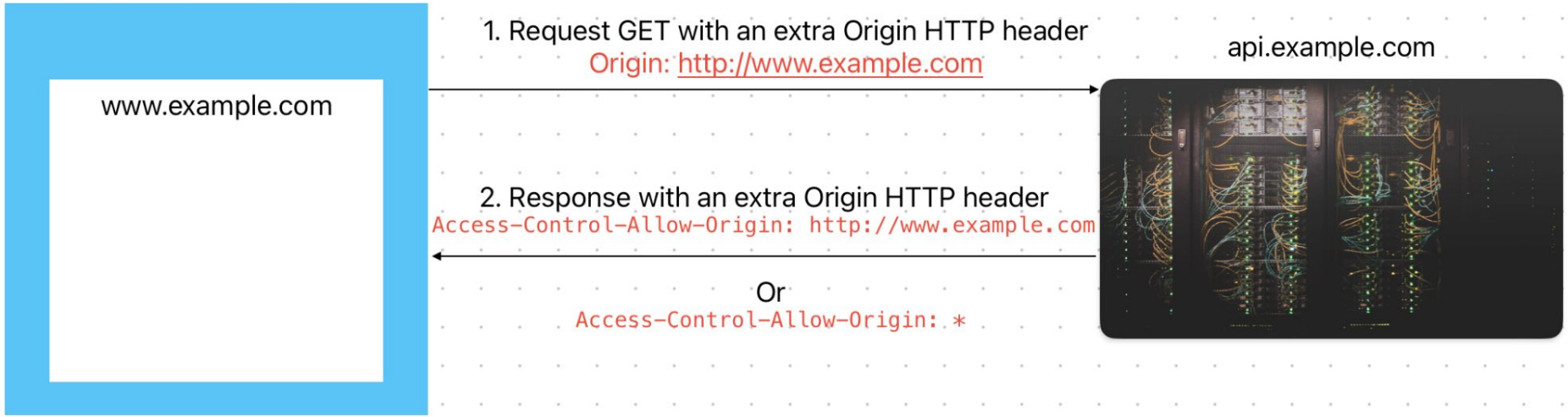
https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

CORS Middleware

Suppose a user visits http://www.example.com and the page attempts a cross-origin request to fetch the user's data from http://api.example.com. A CORS-compatible browser will attempt to make a cross-origin request to api.example.com as follows.

https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

CORS Middleware



https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

CORS Middleware

```
e.Use(middleware.CORS())

DefaultCORSConfig = CORSConfig{
  Skipper:      DefaultSkipper,
  AllowOrigins: []string{"*"},
  AllowMethods: []string{http.MethodGet,
http.MethodHead, http.MethodPut, http.MethodPatch,
http.MethodPost, http.MethodDelete},
}
```

https://echo.labstack.com/middleware/cors/

CORS Middleware

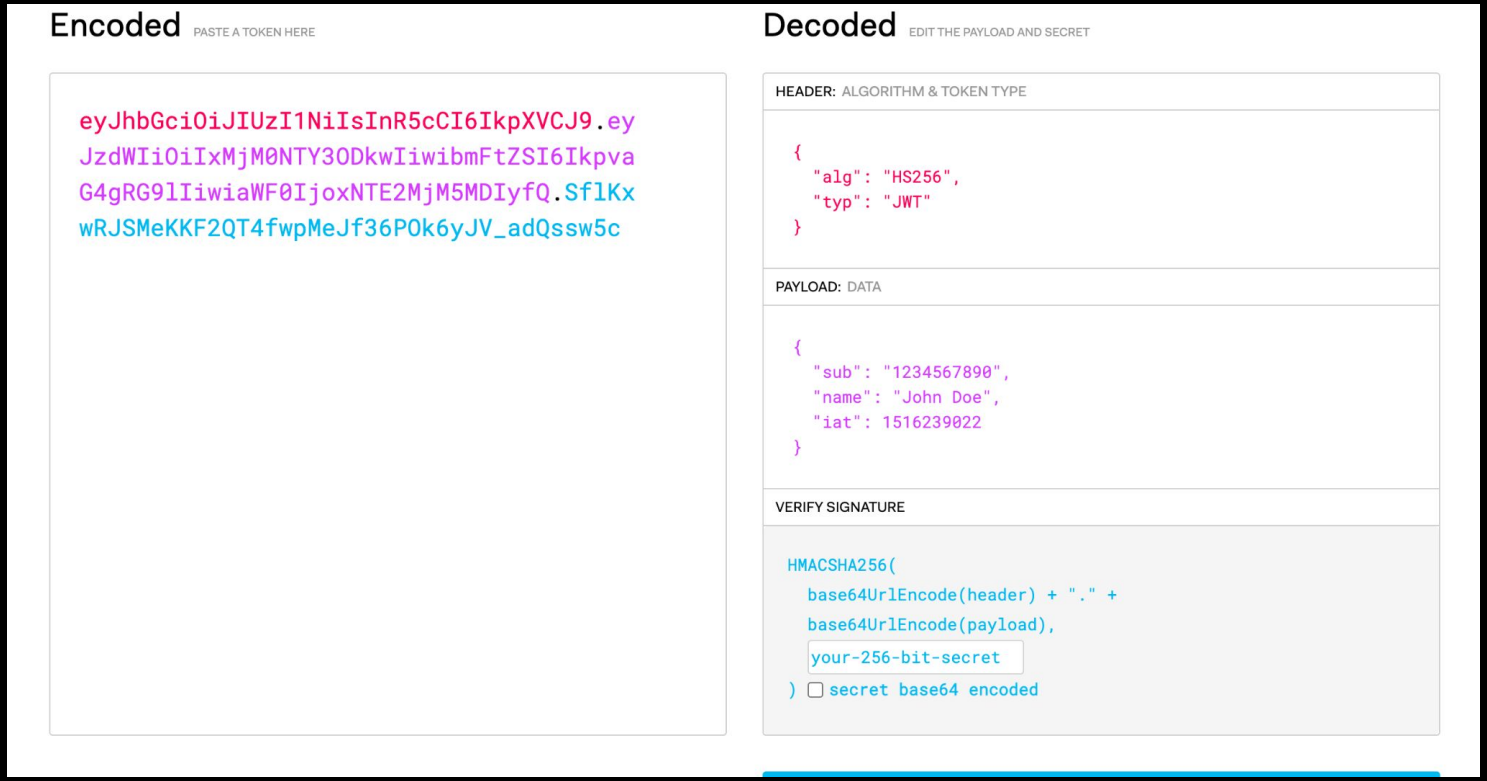
```
e := echo.New()
e.Use(middleware.CORSWithConfig(middleware.CORSConfig{
  AllowOrigins: []string{"https://api.thaibev.com"},
  AllowHeaders: []string{echo.HeaderOrigin,
echo.HeaderContentType, echo.HeaderAccept},
}))
```

https://echo.labstack.com/middleware/cors/

JWT Middleware

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties. JWTs are often used to authenticate users. They can also be used for other purposes, such as sharing information about the user or application. A JWT typically contains a header, a payload, and a signature. The header and payload are base64 encoded JSON strings, and the signature is used to verify the authenticity of the token.

JWT Middleware



JWT Middleware

```
package main

import (
    "github.com/golang-jwt/jwt/v4"
    echojwt "github.com/labstack/echo-jwt/v4"
    "github.com/labstack/echo/v4"
    "github.com/labstack/echo/v4/middleware"
    "net/http"
    "time"
)
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
// jwtCustomClaims are custom claims extending default ones.
// See https://github.com/golang-jwt/jwt for more examples
type jwtCustomClaims struct {
    Name  string `json:"name"`
    Admin bool   `json:"admin"`
    jwt.RegisteredClaims
}
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
func login(c echo.Context) error {
    username := c.FormValue("username")
    password := c.FormValue("password")

    // Throws unauthorized error
    if username != "jon" || password != "shhh!" {
        return echo.ErrUnauthorized
    }
    ...
}
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
func login(c echo.Context) error {
    ...

    // Set custom claims
    claims := &jwtCustomClaims{
        "Jon Snow",
        true,
        jwt.RegisteredClaims{
            ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Hour * 72)),
        },
    }
    ...
}
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
func login(c echo.Context) error {
    ...

    // Create token with claims
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

    // Generate encoded token and send it as response.
    t, err := token.SignedString([]byte("secret"))
    if err != nil {
        return err
    }

    return c.JSON(http.StatusOK, echo.Map{
        "token": t,
    })
}
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
func restricted(c echo.Context) error {
    user := c.Get("user").(*jwt.Token)
    claims := user.Claims.(*jwtCustomClaims)
    name := claims.Name
    return c.String(http.StatusOK, "Welcome "+name+"!")
}

// for c.Get("user") => ContextKey string in Config
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware

```
func main() {
    e := echo.New()
    e.POST("/login", login)
    r := e.Group("/restricted")
    // Configure middleware with the custom claims type
    config := echojwt.Config{
        NewClaimsFunc: func(c echo.Context) jwt.Claims {
            return &jwtCustomClaims{}
        },
        SigningKey: []byte("secret"),
    }
    r.Use(echojwt.WithConfig(config))
    r.GET("", restricted)
    e.Logger.Fatal(e.Start(":1323"))
}
```

https://echo.labstack.com/cookbook/jwt/

JWT Middleware : Custom Middleware

```
func OnlyAdmin(h echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        user := c.Get("user").(*jwt.Token)
        claims := user.Claims.(*jwtCustomClaims)
        if !claims.Admin {
            return echo.NewHTTPError(http.StatusUnauthorized,
"Unauthorized")
        }
        return h(c)
    }
}
```

https://echo.labstack.com/cookbook/jwt/

Logging Middleware

Logger middleware logs the information about each HTTP request.

https://echo.labstack.com/middleware/logger/

Logging Middleware

```
e.Use(middleware.Logger())
```

https://echo.labstack.com/middleware/logger/

Logging Middleware : Custom Configuration

```
e.Use(middleware.LoggerWithConfig(middleware.LoggerConfig{
  Format: "method=${method}, uri=${uri}, status=${status}\n",
}))
```

https://echo.labstack.com/middleware/logger/

Viper

Viper is a complete configuration solution for Go applications. It is designed to work within an application, and can handle all types of configuration needs and formats.

Viper

```go
package config

import (
    "github.com/spf13/viper"
)

var AppConfig Config

type Config struct {
    MongoDbUri        string `mapstructure:"MONGO_DB_URI"`
    MongoDbName       string `mapstructure:"MONGO_DB_NAME"`
    MongoDbUser       string `mapstructure:"MONGO_DB_USER"`
    MongoDbPass       string `mapstructure:"MONGO_DB_PASSWORD"`
}
```

Viper

```
func LoadConfig(path string) (config Config) {
    viper.SetDefault("PORT", "8080")
    viper.SetDefault("MONGO_DB_URI", "mongodb://localhost:27017")
    viper.SetDefault("MONGO_DB_NAME", "newsdb")
    viper.SetDefault("MONGO_DB_USER", "root")
    viper.SetDefault("MONGO_DB_PASSWORD", "password")

    viper.AddConfigPath(path)
    viper.SetConfigName("app")
    viper.SetConfigType("env")
    viper.AutomaticEnv()
    viper.ReadInConfig()
    viper.Unmarshal(&config)
    AppConfig = config
    return AppConfig
}
```

Viper

```
func main() {
    cfg := config.LoadConfig(".")
}
```

Goroutines

A goroutine is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

https://go.dev/tour/concurrency/1

Goroutines

```go
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}
```

https://go.dev/tour/concurrency/1

Goroutines

```
func main() {
    go say("world")
    say("hello")
}
```

https://go.dev/tour/concurrency/1

Channels

Channels are a typed conduit through which you can send and receive values with the channel operator, <-.

https://go.dev/tour/concurrency/2

Channels

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

https://go.dev/tour/concurrency/2

Channels

Like maps and slices, channels must be created before use:

```
ch := make(chan int)
```

https://go.dev/tour/concurrency/2

Graceful Shutdown

```go
package main

import (
    "context"
    "net/http"
    "os"
    "os/signal"
    "time"

    "github.com/labstack/echo/v4"
    "github.com/labstack/gommon/log"
)
```

Graceful Shutdown

```
func main() {
    // Setup
    e := echo.New()
    e.Logger.SetLevel(log.INFO)
    e.GET("/", func(c echo.Context) error {
        time.Sleep(5 * time.Second)
        return c.JSON(http.StatusOK, "OK")
    })

    . . .
}
```

Graceful Shutdown

```
func main() {
    . . .

    // Start server
    go func() {
        if err := e.Start(":1323"); err != nil && err != http.ErrServerClosed {
            e.Logger.Fatal("shutting down the server")
        }
    }()

    . . .
}
```

Graceful Shutdown

```
func main() {
    . . .

    // Wait for interrupt signal to gracefully shutdown the server with a timeout of 10 seconds.
    // Use a buffered channel to avoid missing signals as recommended for signal.Notify
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, os.Interrupt)
    <-quit
    e.Logger.Info("shutdown server. . .")
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()
    if err := e.Shutdown(ctx); err != nil {
        e.Logger.Fatal(err)
    }
    e.Logger.Info("server exiting")
}
```

ldflags

Using `ldflags` to Set Version Information for Go Applications

https://www.digitalocean.com/community/tutorials/using-ldflags-to-set-version-information-for-go-applications

ldflags

```go
package main

import (
    "fmt"
)

var Version = "development"

func main() {
    fmt.Println("Version:\t", Version)
}
```

https://www.digitalocean.com/community/tutorials/using-ldflags-to-set-version-information-for-go-applications

ldflags

```
go build
./app

go build -ldflags="-X
'package_path.variable_name=new_value'"
```

https://www.digitalocean.com/community/tutorials/using-ldflags-to-set-version-information-for-go-applications

ldflags

```
go build -ldflags="-X 'main.Version=v1.0.0'"
```

https://www.digitalocean.com/community/tutorials/using-ldflags-to-set-version-information-for-go-applications