Best Practices for Building Rest APIs with Go

Erhan Yakut / @yakuter

Delivery Hero Tech Hub / 09 November 2021

Biyografi

Ben Kimim?

Erhan YAKUT (yakuter)

İş / Görev

Binalyze isimli **Enterprise Forensics** yazılım şirketinde Senior Software Architect olarak çalışmaktayım.

Programlama Dilleri

Aktif olarak **Go** ile geliştirme yapmakla birlikte uzun yıllar PHP backend developer olarak proje geliştirdim.

Tecrübe/Bilgi

Yaklaşık 15+ yıldır yazılım geliştirme ile ilgilenmekte olup, şu anda işletim sistemleri üzerinde olay sonrası delillerin toplanması için yazılım geliştirmekteyim.

İletişim Bilgisi

Eposta : **yakuter@gmail.com**
Twitter: **@yakuter**

What is REST API?

What is REST API?
An API, or application programming interface, is a set of rules that define how applications or devices can connect to and communicate with each other.

A REST API is an API that conforms to the design principles of the REST, or representational state transfer architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.

Source: https://www.ibm.com/cloud/learn/rest-apis

Let’s continue with survey
https://go.dev/blog/survey2020-results

Checklist

Rest API Checklist

1. Structure
2. Logging
3. Panic and Recover
4. HTTP client and server
5. HTTP Status Codes
6. Routing
7. Middleware (i.e. echo)
    a. CORS
    b. Authentication
    c. Authorization
    d. Rate Limiter
8. JSON issue

Logging

What can I log?

1. Server request and responses
2. Business logic actions
3. Database actions and queries
4. 3rd Party logs

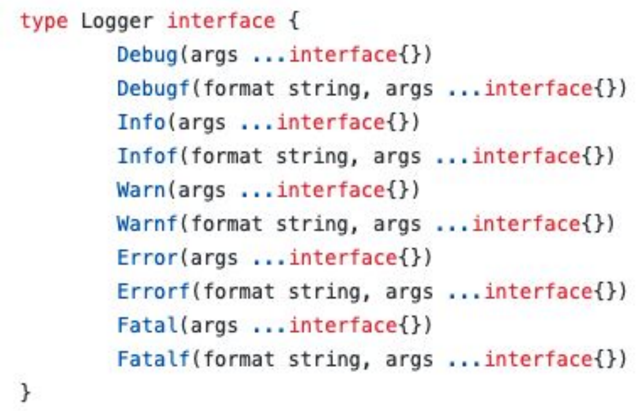
Logging

1. Native Logger (i.e. link)
   https://pkg.go.dev/log

2. Logrus
   https://github.com/sirupsen/logrus

3. Zerolog
   https://github.com/rs/zerolog

4. Uber Zap
   https://github.com/uber-go/zap



```go
type Logger interface {
        Debug(args ...interface{})
        Debugf(format string, args ...interface{})
        Info(args ...interface{})
        Infof(format string, args ...interface{})
        Warn(args ...interface{})
        Warnf(format string, args ...interface{})
        Error(args ...interface{})
        Errorf(format string, args ...interface{})
        Fatal(args ...interface{})
        Fatalf(format string, args ...interface{})
}
```

Panic and Recover

Panic

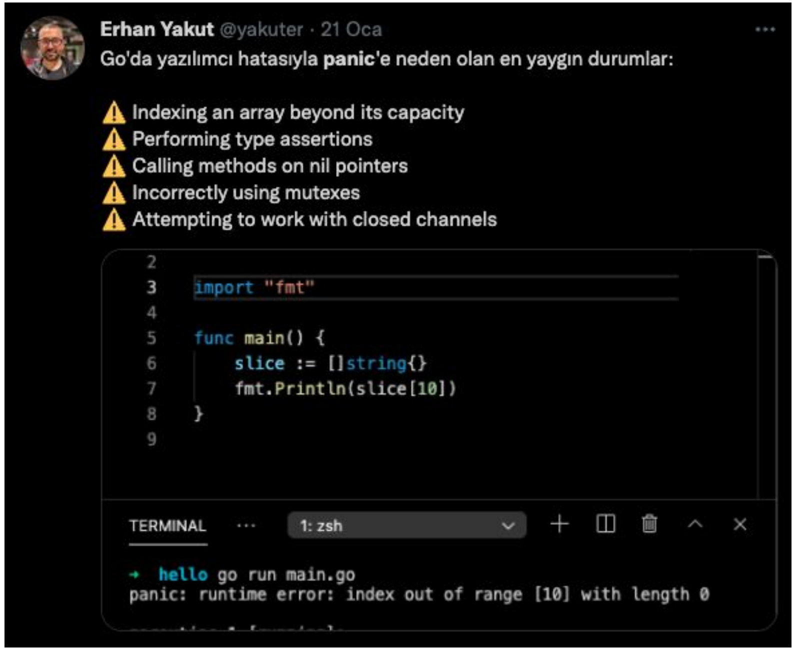
Panic is a built-in function that stops the ordinary flow of control and begins panicking. When the function F calls panic, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller...

Recover

Recover is a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions. During normal execution, a call to recover will return nil and have no other effect. If the current goroutine is panicking, a call to recover will capture the value given to panic and resume normal execution.

Source:

https://go.dev/blog/defer-panic-and-recover
https://www.kirsle.net/globally-recover-panics-in-go
https://play.golang.org/p/gF1jCfKFRSB



Erhan Yakut @yakuter · 21 Oca

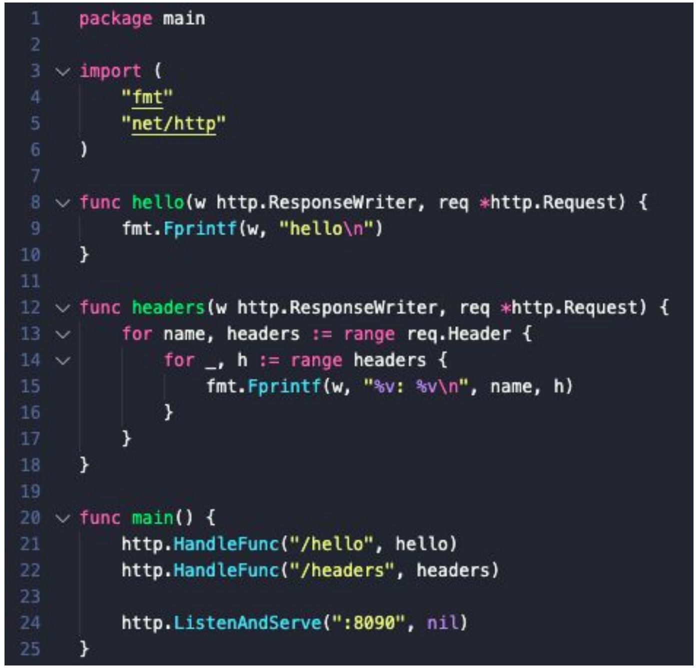
Go'da yazılımcı hatasıyla **panic**'e neden olan en yaygın durumlar:

⚠ Indexing an array beyond its capacity
⚠ Performing type assertions
⚠ Calling methods on nil pointers
⚠ Incorrectly using mutexes
⚠ Attempting to work with closed channels

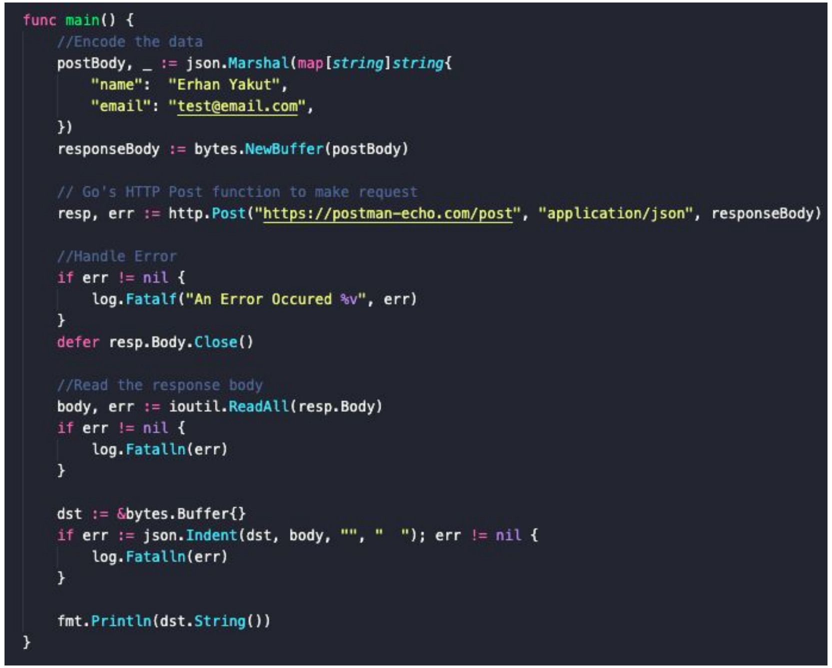
```go

import "fmt"

func main() {
    slice := []string{}
    fmt.Println(slice[10])
}

```

```
→ hello go run main.go
panic: runtime error: index out of range [10] with length 0
```

HTTP Client and Server



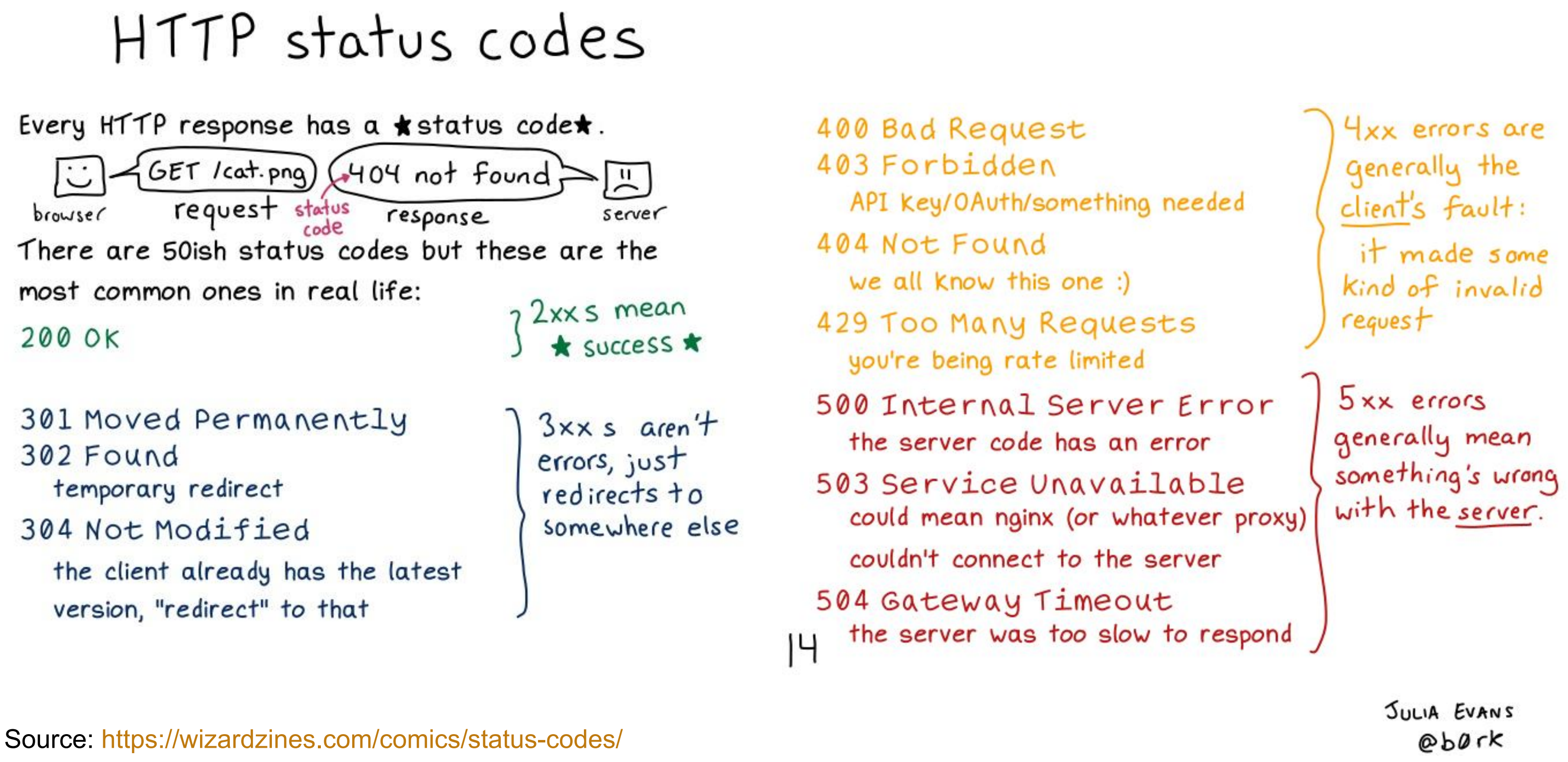
```go
package main

import (
	"fmt"
	"net/http"
)

func hello(w http.ResponseWriter, req *http.Request) {
	fmt.Fprintf(w, "hello\n")
}

func headers(w http.ResponseWriter, req *http.Request) {
	for name, headers := range req.Header {
		for _, h := range headers {
			fmt.Fprintf(w, "%v: %v\n", name, h)
		}
	}
}

func main() {
	http.HandleFunc("/hello", hello)
	http.HandleFunc("/headers", headers)

	http.ListenAndServe(":8090", nil)
}
```



```go
func main() {
	//Encode the data
	postBody, _ := json.Marshal(map[string]string{
		"name":  "Erhan Yakut",
		"email": "test@email.com",
	})
	responseBody := bytes.NewBuffer(postBody)

	// Go's HTTP Post function to make request
	resp, err := http.Post("https://postman-echo.com/post", "application/json", responseBody)

	//Handle Error
	if err != nil {
		log.Fatalf("An Error Occured %v", err)
	}
	defer resp.Body.Close()

	//Read the response body
	body, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		log.Fatalln(err)
	}

	dst := &bytes.Buffer{}
	if err := json.Indent(dst, body, "", "  "); err != nil {
		log.Fatalln(err)
	}

	fmt.Println(dst.String())
}
```

HTTP Status Codes

Source: https://wizardzines.com/comics/status-codes/



HTTP status codes

Every HTTP response has a ★status code★.

There are 50ish status codes but these are the most common ones in real life:

200 OK

2xxs mean ★ success ★

301 Moved Permanently

302 Found
temporary redirect

304 Not Modified
the client already has the latest version, "redirect" to that

3xxs aren't errors, just redirects to somewhere else

400 Bad Request

403 Forbidden
API key/OAuth/something needed

404 Not Found
we all know this one :)

429 Too Many Requests
you're being rate limited

4xx errors are generally the client's fault: it made some kind of invalid request

500 Internal Server Error
the server code has an error

503 Service Unavailable
could mean nginx (or whatever proxy) couldn't connect to the server

504 Gateway Timeout
the server was too slow to respond

5xx errors generally mean something's wrong with the server.

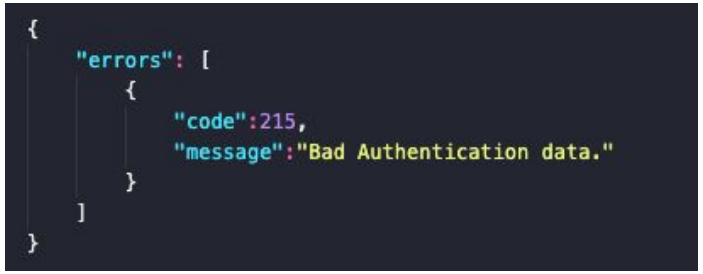
14

JULIA EVANS
@b0rk

Source: https://wizardzines.com/comics/status-codes/

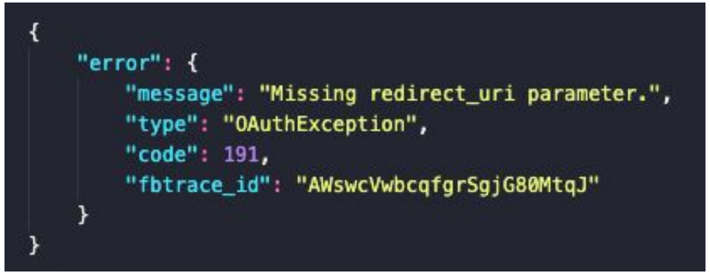
Error Responses

Twitter



```
{
    "errors": [
        {
            "code":215,
            "message":"Bad Authentication data."
        }
    ]
}
```

Facebook



```
{
    "error": {
        "message": "Missing redirect_uri parameter.",
        "type": "OAuthException",
        "code": 191,
        "fbtrace_id": "AWswcVwbcqfgrSgjG80MtqJ"
    }
}
```

Routing

Routers

1. Native HTTP ServeMux
   https://pkg.go.dev/net/http#ServeMux

2. Gorilla Mux Router
   https://github.com/gorilla/mux

3. Httprouter
   https://github.com/julienschmidt/httprouter

4. Fasthttp Router
   https://github.com/valyala/fasthttp

5. Chi
   https://github.com/go-chi/chi

Source:
https://www.alexedwards.net/blog/which-go-router-should-i-use
https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
https://astaxie.gitbooks.io/build-web-application-with-golang/content/en/13.2.html

- By **supports method-based routing** I mean that the router makes it easy to dispatch a HTTP request to different handlers based on the request method (`"GET"`, `"POST"`, etc).
- By **supports variables in URL paths** I mean that the router makes it easy to declare routes like `/movies/{id}` where `{id}` is a dynamic value in the URL path.
- By **supports regexp route patterns** I mean that the router makes it easy to declare routes like `/movies/{[a-z-]+}` where `[a-z-]+` is a required regexp match in the URL path.
- By **supports host-based routes** I mean that the router makes it easy to dispatch a HTTP request to different handlers based on the URL *host* (like `www.example.com`) rather than just the URL *path*.
- By **supports custom routing rules** I mean that the router makes it easy to add custom rules for routing requests (such as routing to different handlers based on IP address, or the value in an `Authorization` header).
- By **conflicting routes** I mean when you register two (or more) route patterns that potentially match the same request URL path. For example, if you register the routes `/blog/{slug}` and `/blog/new` then a HTTP request with the path `/blog/new` matches *both* these routes.

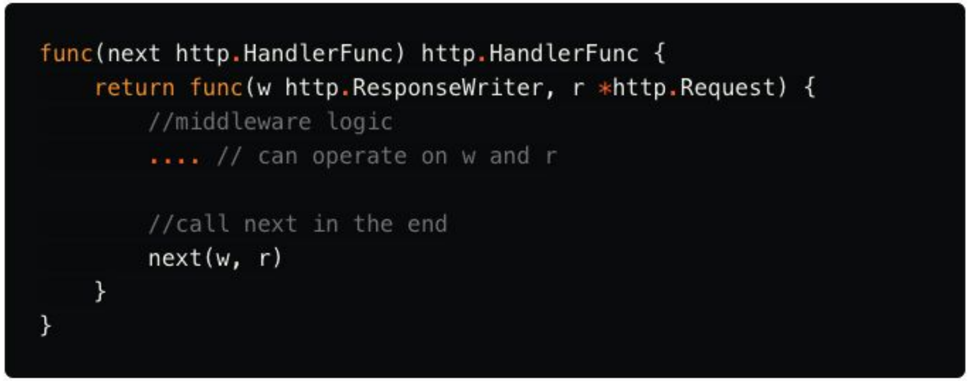
Middleware

Middlewares

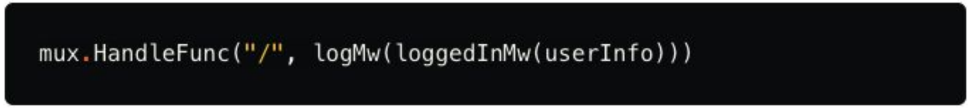
1. Native HTTP HandlerFunc
   https://pkg.go.dev/net/http#HandlerFunc

2. Alice
   https://github.com/justinas/alice

3. Negroni
   https://github.com/urfave/negroni



```go
func(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        //middleware logic
        .... // can operate on w and r

        //call next in the end
        next(w, r)
    }
}
```



```go
mux.HandleFunc("/", logMw(loggedInMw(userInfo)))
```

JSON

JSON Parsers

1. Native Parser
   https://pkg.go.dev/encoding/json

2. Easy JSON
   https://github.com/mailru/easyjson

3. Jsonparser
   https://github.com/buger/jsonparser

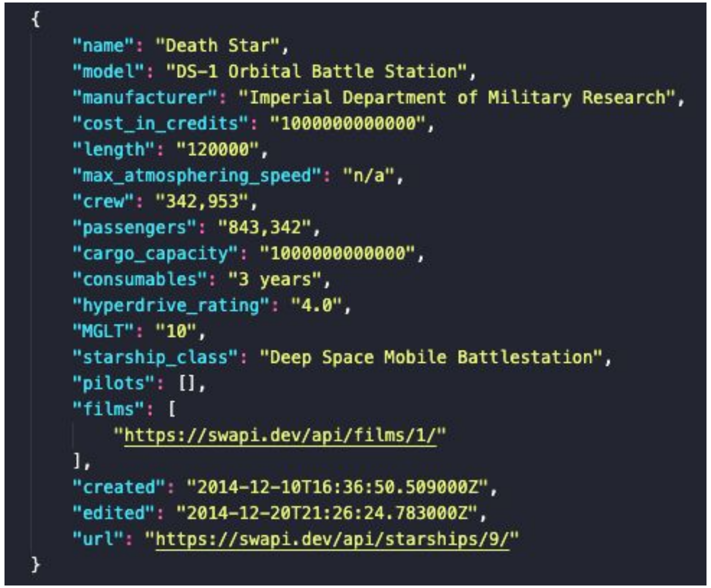
4. Jsoniter
   https://github.com/json-iterator/go

JSON&Struct Conversion

https://mholt.github.io/json-to-go/

Validation

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



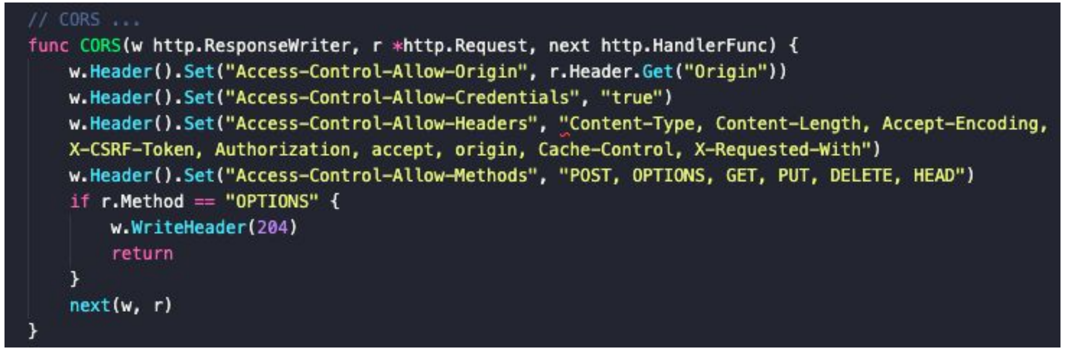
```json
{
    "name": "Death Star",
    "model": "DS-1 Orbital Battle Station",
    "manufacturer": "Imperial Department of Military Research",
    "cost_in_credits": "1000000000000",
    "length": "120000",
    "max_atmosphering_speed": "n/a",
    "crew": "342,953",
    "passengers": "843,342",
    "cargo_capacity": "1000000000000",
    "consumables": "3 years",
    "hyperdrive_rating": "4.0",
    "MGLT": "10",
    "starship_class": "Deep Space Mobile Battlestation",
    "pilots": [],
    "films": [
        "https://swapi.dev/api/films/1/"
    ],
    "created": "2014-12-10T16:36:50.509000Z",
    "edited": "2014-12-20T21:26:24.783000Z",
    "url": "https://swapi.dev/api/starships/9/"
}
```

CORS Middleware

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.

CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.



```go
// CORS ...
func CORS(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
    w.Header().Set("Access-Control-Allow-Origin", r.Header.Get("Origin"))
    w.Header().Set("Access-Control-Allow-Credentials", "true")
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type, Content-Length, Accept-Encoding,
    X-CSRF-Token, Authorization, accept, origin, Cache-Control, X-Requested-With")
    w.Header().Set("Access-Control-Allow-Methods", "POST, OPTIONS, GET, PUT, DELETE, HEAD")
    if r.Method == "OPTIONS" {
        w.WriteHeader(204)
        return
    }
    next(w, r)
}
```

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

Thank You