Introduction à Go – TD2: HTTP

Franklin ”Snaipe” Mathieu

2023

Dans ce TD, nous allons, pas à pas, voir comment créer un service HTTP pour faire du processing JSON.

1 Faire un service HTTP avec echo

1.1 Hello, HTTP!

Pour faire notre service, nous allons utiliser `github.com/labstack/echo`.

Echo est une bibliothèque conçue pour écrire facilement des services HTTP.

Commençons par créer un dossier de projet:

```
$ go mod init example.com/http-service
$ go get github.com/labstack/echo/v4
```

Créez un fichier main.go:

```
package main

import (
        "fmt"
        "net/http"

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

func main() {
        // Créée une instance d'Echo
        e := echo.New()

        // Associe une fonction qui retourne un HTTP 200 OK avec en corps
        // "Hello, World!\n" à la racine du service
        e.GET("/", func(c echo.Context) error {
                return c.String(http.StatusOK, "Hello, World!\n")
        })

        // Démarre le serveur et bloque jusqu'à ce qu'il finisse
        if err := e.Start(":1323"); err != nil {
                fmt.Println(err)
        }
}
```

Lançons ce serveur:

```
$ go run .

   ____    __
  / __/___/ /  ___
 / _// __/ _ \/ _ \
/___/\__/_//_/\___/ v4.11.2
High performance, minimalist Go web framework
https://echo.labstack.com
____________________________________O/_______
                                    O\
⇨ http server started on [::]:1323
```

Dans un autre terminal, vous pouvez tester ce service:

```
$ curl localhost:1323
Hello, World!
```

1.2 Un service compteur

Nous allons changer le service pour traquer un compteur au sein du programme, de sorte à ce que ces chemins soient implémentés:

- `GET /counter` retourne la valeur du compteur
- `PUT /counter` assigne la valeur du compteur
- `PATCH /counter` incrémente (ou décrémente) le compteur par la valeur passée

Pour faire ça, nous allons créer un nouveau type pour représenter notre service.

Faites un nouveau fichier counter.go:

```
package main

type CounterService struct {
        value int
}
```

Notre type CounterService contiendra l'état du service, et donc la valeur courante du compteur. Ajoutons une méthode pour retourner la valeur:

```
func (svc *CounterService) GetCounter(c echo.Context) error {
        var result struct {
                Counter int `json:"counter"`
        }
        result.Counter = svc.value

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

... et une méthode pour l'assigner:

```
func (svc *CounterService) PutCounter(c echo.Context) error {
        var params struct {
                Counter int `json:"counter"`
        }
        c.Bind(&params) // parse le corps de la requête HTTP dans `params`

        svc.value = params.Counter

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

Puis ajoutons dans le `main()` notre service:

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

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

        var svc CounterService
        e.GET("/counter", svc.GetCounter)
        e.PUT("/counter", svc.PutCounter)

        if err := e.Start(":1323"); err != nil {
                fmt.Println(err)
        }
}
```

PATCH est laissé en exercice au lecteur; implémentez-le, nous en aurons besoin.

Vous pouvez relancer votre service avec ‘go run .‘, puis tester le compteur avec curl:

```
$ curl localhost:1323/counter
{"counter":0}

$ curl -X PUT \
>  -H 'Content-Type: application/json' \
>  -d '{"counter":42}' \
>  localhost:1323/counter
{"counter":42}

$ curl localhost:1323/counter
{"counter":42}

$ curl -X PUT \
>  -H 'Content-Type: application/json' \
>  -d '{"counter":0}' \
>  localhost:1323/counter
{"counter":0}
```

Malheureusement, ce service a un problème; il a une race condition. Pour la voir, lançons 1000 commandes curl concurrentiellement:

```
$ for i in $(seq 1 1000); do
>   curl -sS -X PATCH \
>   -H 'Content-Type: application/json' \
>   -d '{"counter":1}' \
>   localhost:1323/counter &
> done; wait

$ curl localhost:1323/counter
{"counter":979}
```

La valeur finale du compteur aurait dû être 1000, mais elle ne l’est pas.

Go vous donne des outils pour débugger le problème. En particulier, go run (et go test) peut être lancé avec un ”race detector”:

```
$ go run -race .
```

Si vous répétez les 1000 commandes curl, vous verrez que le programme affichera des warnings sur la data race du compteur.

Ce problème arrive parce que chaque requête HTTP est exécutée dans sa propre goroutine, et donc sans synchronisation, chaque goroutine tente d’incrémenter de manière non-atomique le compteur.

Tentez de régler le problème. Deux voies sont possibles: utiliser de la synchronisation (regardez le package `sync`), ou utiliser une variable atomique (regardez le package `sync/atomic`).

2 Faire des requêtes HTTP

Nous avons vu la partie serveur, voyons maintenant la partie client!

Dans un dossier à part, faites un nouveau main.go:

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
)

func main() {
    // Fais une requête HTTP GET http://localhost:1323/counter
    resp, err := http.Get("http://localhost:1323/counter")
    if err != nil {
        log.Fatal(err)
    }

    // Quand la fonction se termine, on ferme le flux de réponse
    defer resp.Body.Close()

    var result struct {
        Counter int `json:"counter"`
    }

    // On décode le json du le corps de réponse dans la variable result
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        log.Fatal(err)
    }

    fmt.Println("counter:", result.Counter)
}
```

Ce programme effectue une requête GET pour récupérer la valeur du compteur de votre service, puis l'affiche.

`http.Get` est une forme plus courte de:

```
req, err := http.NewRequest("GET", "http://localhost:1323/counter", nil)
if err != nil {
    log.Fatal(err)
}

resp, err := http.DefaultClient.Do(req)
if err != nil {
    log.Fatal(err)
}
```

Essayez de modifier le programme pour assigner une valeur au compteur via une requête PUT (indice: utilisez la fonction http.DefaultClient.Do et http.NewRequest).

Pour sérialiser une structure dans un buffer:

```
var val struct {
    Counter int `json:"counter"`
}

var out bytes.Buffer
json.NewEncoder(&out).Encode(val)
```

La variable `out` peut être utilisée en tant que corps de requête dans le 3e argument de http.NewRequest.

3 Tout ensemble!

Nous avons créé un service et un client séparément; regroupons tout dans un seul programme.

Faites un binaire nommé ”counter” qui implémente les commandes suivantes:

- `counter serve [host:port]`: lance le service http du service counter, avec en option le host:port sur lequel démarrer le serveur
- `counter get host:port`: récupère et affiche la valeur du compteur en parlant au serveur http sur host:port
- `counter set host:port N`: assigne la valeur du compteur à N en parlant au serveur http sur host:port
- `counter add host:port N`: ajoute N à la valeur du compteur en parlant au serveur http sur host:port

N’oubliez pas d’utiliser github.com/alecthomas/kong pour votre CLI.