TD 3
Golang pour DevOps

1 Concurrence simple, et interfaces

1.1. Considérez le code suivant:

```
package main

import (
    "fmt"
)

func main() {
    go     maFonction()
    fmt.Println("Fin du programme")
}

func maFonction() {
    fmt.Println("j'ai fini !")
}
```

Modifiez-le pour que "maFonction()" ait toujours le temps de s'exécuter complètement avant la fin du programme.
Pour cela, utilisez un WaitGroup:

```
https://golang.org/pkg/sync/#WaitGroup
```

Vous pouvez modifier la signature de la fonction "maFonction()" (c'est même encouragé).

2 Channels, select et HTTP

On va utiliser dans cette partie les concepts vus en cours aujourd'hui de channels et de select.
On va également reprendre ce qu'on a vu dans le TD2 concernant le HTTP, mais en se plaçant cette fois du côté d'un client, et non plus d'un serveur.

2.1. Définissez un type "Reponse", qui est une struct comprenant deux champs: respText de type string, et err de type error.

2.2. Créez dans, votre fonction main(), deux channels qui transportent des types "Reponse".

2.3. Écrivez une fonction "callServer" qui prend deux arguments:

- une adresse, sous forme de chaine de caractères
- un channel qui transporte des "Reponse"

2.4. Dans votre fonction callServer, réalisez une requête HTTP vers le serveur dont l'adresse a été passée en argument.
Vous utiliserez pour cela la fonction "Get" du package http.

2.5. Toujours dans callServer, gérez les réponses possibles du serveur:

(a) Si l'appel à Get retourne une erreur, créez un objet "Reponse" et remplissez le champ "err". Envoyez cet objet dans le channel passé en argument et arrêtez l'exécution de la fonction avec "return".

(b) Si la réponse que vous obtenez présente un status code HTTP qui est différent de 200, créez à nouveau un objet Reponse, et peuplez le champ "err" de celui-ci avec (c'est à mettre sur une seule ligne):

```
errors.New("Le code retourné par le serveur indique une
    erreur: " + strconv.Itoa(resp.StatusCode))
```

(c) Lisez le corps de la réponse HTTP en utilisant le code suivant (adaptez avec les noms de variables que vous aurez utilisés):

```
body, err := ioutil.ReadAll(resp.Body)
```

(d) Pensez à fermer le body de la réponse http avec un "defer".

(e) Si la lecture du code de la réponse retourne une erreur, créez un objet Reponse, mettez cette erreur dans le champ err et passez la Reponse dans le channel.

(f) Si tout s'est passé comme prévu, envoyez dans le channel un Reponse en peuplant le champ "respText" avec le contenu du body de la réponse du serveur.
Pensez bien que ioutil.ReadAll qui vous a servi à lire le body retourne []byte et non pas string. Il faut faire un "cast".

2.6. On retourne à présent dans notre fontion main. On va faire les appels au serveur.
Autrement dit, main() va utiliser votre fonction callServer.
À la suite des déclarations des channels faites précédemment, ajoutez deux appels à callServer qui respectivement:

- appelle l'adresse: "http://localhost:8000/?id=id1" et prend le channel 1 en argument
- appelle l'adresse: "http://localhost:8000/?id=id3" et prend le channel 2 en argument

2.7. À présent, lancez vos appels au serveur dans une goroutine chacun, avec:

```
go callServer(...) // Remplissez avec les bons arguments
```

Que se passe-t-il si vous lancez votre programme comme cela ? Pourquoi ?

2.8. Pour attendre que les fonctions terminent sans utiliser de waitgroup, on va lire sur les channels.
À la suite de vos appels au serveur, lisez les informations qui sont sur vos channels en utilisant:

```
<-ch1
<-ch2
```

2.9. Écrivez maintenant un select, qui lit pour chaque "case" sur un channel différent.
Par ce biais, vous pourrez voir quel endpoint est le plus rapide à répondre puisque select va prendre le premier résultat mis à sa disposition.
Quel est l'endpoint le plus rapide ?
Quelle est la réponse de cet endpoint ?