# Golang

Une introduction pour DevOps - Part 2

# Rappels

- On a deux catégories de types en Go : value et headers
- Un type value décrit directement la valeur correspondante (un bool est bien un boolean, et pas une adresse vers un boolean)
- Un type header contient des informations sur les valeurs qu'il décrit. On peut bien sûr récupérer

# Rappels

- Exemple de type header : les slices (trois valeurs: pointeur vers le premier élément, longueur, capacité)
- Les zero values : ce sont les valeurs par défaut des types en Go.

# Rappels pointeurs

- Les pointeurs sont des adresses mémoire
- Ils désignent un emplacement, où trouver "autre chose"
- Cela peut être une valeur, ou un autre pointeur etc...

Mais alors pourquoi utiliser des pointeurs ?

# Pourquoi ?

Qu'affiche ce code ?

```
package main
import (
  "fmt"
)
func main() {
  ex:= Example{age: 10}
  modifAge(ex, 43)
  fmt.Println(ex)
}

type Example struct{
  age int
}

func modifAge (ex Example, age int){
```

{10}

# Pourquoi ?

- Go est "pass by value"
  - C'est à dire que quand on passe une variable à une fonction par exemple, unecopie est faite, et c'est cette copie qui est utilisée dans la fonction.
- Comment "réparer" l'exemple précédent ? => Avec des pointeurs !

```go
package main
import (
  "fmt"
)
func main() {
  ex:= Example{age:10}
  modifAge(&ex, 43)

  fmt.Println(ex)
}

type Example struct{
  age int
}
```

# Les interfaces

- C'est quoi une interface ?
- Un contrat :
  - Une liste de méthodes que doit respecter une entité qui l'implémente.
  - Si l'entité implémente ces méthodes, elle est

# Les interfaces

- Une interface permet de grouper des types concrets par fonctionnalité
- Les interfaces peuvent aider pour tester le code
- Une interface est vérifée implicitement : si toutes les méthodes d'une interface sontimplémentées pour un type, alors ce type vérifie l'interface

# Les interfaces

- Zero value : que se passe-t-il si on execute ce programme ?

```
package main

func main() {
  var ifce IfceTest
  ifce.test()
}

type IfceTest interface {
  test()
}
```

- PANIC

# Les interfaces

- La zero value d'une interface est un pointeur nil
- Un pointeur nil, pour rappel, c'est une absence de pointeur.
- Appeler une méthode sur un pointeur nil, c'est appeler une méthode sur "rien" =>Panic

# Les interfaces

- Comment initialiser l'interface dans notre exemple précédent ?

```
package main

func main()
{
  var ifce IfceTest
  ifce = Example{}
  ifce.test()

}

type IfceTest interface {
  test()
}

type Example struct{}
```

# Les interfaces

- Si on a besoin de récupérer le type qui se cache sous une interface, au runtime ?

=> C'est possible avec un switch :

```
func printType(i interface{}) {
  switch v := i.(type) {
    case int:
      fmt.Println("The type is int !")
    case string:
      fmt.Println("The type is string !")
    default:
      fmt.Printf("I don't know about type %T!\n", v)
    }
}
```

# Les interfaces

- Comment vérifier qu'une interface "contient" bien un type particulier (sans passer par le switch) ?

```
package main

import
(
  "log"
)

func main()
{
  var ifce Ifce
  ifce = Example{}
  _, ok := ifce.(Example)
  if !ok {
    log.Fatal("Pas le type attendu !")
  }
```

# Les interfaces

- En écriture condensée :

```
package main

import (
  "log"

)

func main() {
  var ifce Ifce
  ifce = Example{}

  if _, ok := ifce.(Example);!ok {
    log.Fatal("Pas le type attendu !")
  }
}
```

# Parsing JSON

- JSON est très utilisé dans les APIs de nos jours, et existe aussi dans les logs
- Parsing intégré dans la librairie standard
- Parsing se fait par annotations

# Parsing JSON

- Exemple de données au format JSON :

```
{
   "menu":{
      "id":"file",
      "value":"File",
      "popup":{
         "menuitem":[
            { "value":"New", "onclick":"CreateNewDoc()" },
            { "value":"Open", "onclick":"OpenDoc()" },
            { "value":"Close", "onclick":"CloseDoc()"}
         ]
      }
   }
}
```

# Parsing JSON

- Quels avantages ?
  - Simple à manipuler pour un programmeur
  - Lisible par un humain, léger pour les machines
  - "Facile à apprendre" parce que la syntaxe n'est pas extensible

# Parsing JSON

- Quels inconvénients ?
  - Syntaxe non extensible (contrairement à du XML par exemple)
  - Le typage limité affaiblit la sécurité
  - On ne peut pas toujours commenter du JSON (dépend du parser)

# Parsing JSON

- Comment générer du JSON ?
  - Créer une struct qui correspond aux données que l'on veut en sortie
  - Utiliser des annotations si nécessaire
  - Utiliser https://golang.org/pkg/encoding/json/#Marshal

- Qu'affiche ce programme ?

```
package main

import (
    "encoding/json"
    "fmt"
)
func main() {
  group: = ColorGroup {
    ID: 1,
    Name: "Reds",
    Colors: []string {"Crimson", "Red", "Ruby", "Maroon"},
  }
  b, err: = json.Marshal(group)
  if err != nil {
    fmt.Println("error:", err)
```

```
{"ID":1,"Name":"Reds","Colors":["Crimson","Red","Ruby","Maroon"]}
```

# Parsing JSON

- Comment déserialiser du JSON ?
  - Créer une struct qui correspond aux données que l'on veut lire
  - Utiliser des annotations si nécessaire
  - Utiliser https://golang.org/pkg/encoding/json/#Unmarshal

```go
package main

import (
  "encoding/json"
  "fmt"
)


func main() {
  var jsonBlob = []byte(
    `[
        {"Name": "Platypus", "Order": "Monotremata"},
        {"Name": "Quoll", "Order": "Dasyuromorphia"}
    ]`)
```

# Parsing JSON

Importants : seuls les champs exportés seront serialisés/déserialisés.

# Parsing JSON

## Les annotations

- Les annotations permettent de redéfinir les noms des champs entre JSON et struct.
- Exemple :

```
type Animal struct {
  Species string `json:"Name"`
  Od string `json:"Order"`
}
```

- On peut aussi spécifier que l'on ne veut pas que des champs vides soient ajoutésdans les JSON générés avec l'annotation suivante :

```
type Animal struct {
  Species string `json:",omitempty"`
  Order string
}
```

# Qu'affiche ce programme ?

```
package main
import (
    "encoding/json"
    "fmt"
)

func main() {

  type ColorGroup struct {
    ID int
    Name string
    Colors []string `json:",omitempty"`
  }
  group: = ColorGroup {
    ID: 1,
    Name: "Reds",
```

```
{"ID":1,"Name":"Reds"}
```

# Les maps !

- C'est un header type
- Syntaxe

```
map[Type_des_clés]Type_des_values
```

- Initialiser avec des valeurs :

```
m := map[string]int{"bob": 5}
```

# Les maps

- Récuperer des valeurs (attention, si la clé est absente, on récupère la zero valuedu type des valeurs !) :

```
valeur := m["bob"]
```

- Bonus : Récupérer une valeur et faire quelque chose si et seulement si la clé était présente :

```
if val, ok := m["bob"]; ok {
  //do something here
}
```