Go

Cours

Cours

Introduction

Qui a fait le Go ?

Go (ou Golang) est un langage de programmation open source assez jeune.

Il a été développé en 2007 par Robert Griesemer, Rob Pike et Ken Thompson qui travaillent aujourd'hui chez Google.

Untitled.png

Le langage Go est officiellement lancé en novembre 2009.

Pourquoi le Go ?

> "*Chez Google, nous pensons que la programmation devrait être rapide, productive et surtout, fun. C'est pourquoi nous sommes ravis de proposer ce nouveau langage de programmation expérimental. Les opérations de compilation sont presque instantanées,et le code compilé propose une vitesse de fonctionnement proche de celle du C*".

> Première phrase du site golang.org "*Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.*"

Pour résumer : Le langage de programmation Go rime avec efficacité et simplicité.

> Énormément de concepts de l'époque sont redécouvert aujourd'hui

Ceci est un ...

Untitled 1.png

Geomyidae - Wikipédia

Héritage très fort des langages des années 70'

Go est syntaxiquement similaire au langage C mais contrairement au C, il possède une sécurité de la mémoire avec un Garbage Collector.

Go est souvent comparé au langage Python car tous les 2 se veulent très simples syntaxiquement.

Les avantages du langage Go

- Une meilleure protection de la mémoire grâce à son Garbage Collector qui permet une gestion automatique de la mémoire.
- Profite de la puissance de calcul des processeurs les plus robustes du marché (processeurs multi-cœurs).
- Un code maintenable
- Possibilité de faire du typage dynamique et intègre de nombreux types avancés tels que les mappages clé-valeur (dictionnaires).
- Possède une si riche bibliothèque standard, qu'il est même tout à fait possible de concevoir des programmes écrit avec le langage Go sans aucune dépendance externe.
- Possède un **temps de compilation rapide** et intègre aussi un système de build beaucoup moins compliqué que celui de la plupart des langages de compilés (RIP le C et son makefile de l'enfer !!!).
- Au niveau de la **portabilité** il est possible de compiler son code pour une large gamme de systèmes d'exploitation et de plateformes matérielles (Windows, Linux, MAC OS, Android, IOS).

Utilisation du langage Go

On retrouve le langage Go dans les domaines suivants (liste non exhaustive) :

- Serveurs
- Web
- Systèmes embarqués
- IOT (Internet Of Things)
- Android
- IOS
- Jeux-vidéos

- etc ...

Des entreprises utilisant Go :

- CloudFlare
- DropBox
- Docker □□
- Nokia
- OVH
- Youtube
- SoundCloud
- Github
- Netflix
- etc...

Principles features du Go

Pour résumer, le Go c'est :

- Langage statique & compilé (Java, C, C++, ...)
- Syntaxe proche du C
- Garbage Collector ➡Gestion de la mémoire automatique □□
- Multi-CPU et parallelism dans le langage (sans avoir recours à des librairies externes)
- Un seul binaire, aucune dépendance !
- Multi-plateforme (Windows, macOS, Linux, arm, ...)

Par ici l'installation !

The Go Programming Language

Pour l'IDE

- VSCode (avec les extensions suggérées)
- Jetbrains Goland
- Atom (avec le plugin https://atom.io/packages/go-plus)
- Vim-go (oui oui...)
- SublimeText (avec le plugin https://github.com/DisposaBoy/GoSublime)

Pour en trouver d'autres :

IDEsAndTextEditorPlugins · golang/go Wiki

Cours

Les bases

Clean architecture Go

Untitled.png

Le playground

https://play.golang.org/

Bonjour monde

```
package main

import "fmt"

func main() {
  fmt.Println("Hello, World!")
}
```

Les types

Langage fortement typé, avec possibilités de faire de l'inférence.

On peut déclarer une variable avec un type de plusieurs façons.

Les types basiques

```
bool             // true / false

string           // "Hello", "Goodbye"

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr
□□□□ // 10, 20, 42255

byte             // 01001110 ==> stockage sur un octet (alias de uint8)

rune             // alias de int32

float32 float64  // 3.14 ==> équivalent à float et double
```

Déclaration d'une variable

```
var number int = 20
var age int          // = 0

var firstname string = "Nou"
var lastname string  // = ""
```

Déclaration par inférence

```
number := 10

firstname := "Nono"
```

> On peut aussi utiliser les décalages de bits ! (>> , <<).
> Non ça n'est pas un smiley

Cast

```
var n int = 42
f := float64(n) + .42
fmt.Printf("float=%f\n", f)
```

Cours

Tests & Boucles

Les combinaisons et opérateurs booléens

Comparaisons

- `==`
- `!=`
- `<`
- `>`
- `>=`
- `>=`

Opérateurs booléens

- `&&`
- `||`
- `!`

If

```
age := 10
if age > 10 {
	// something
}
```

Conditions alternatives

```
age := 10
if age > 10 {
	// something
} else if a > 5 {
    // something else
} else {
    // something else else
}
```

Switch

> Hé Il n'y a pas de break dans les switch en go

```
age := 10
switch age {
case 10:
	// code cas 10
case 5 :
	// code cas 5
default :
	// code par défaut (facultatif)
}
```

On peut cumuler les valeurs au sein d'un même `case` !

```
age := 10
switch age {
	case 10,8,6:
		// code cas 10 ou 8 ou 6
	case 5 :
		// code cas 5
	default :
		// code par défaut (facultatif)
}
```

Et même mieux encore...

```
hour := 10
fmt.Printf("Il est %dh\n", hour)
switch {
	case hour < 12:
		fmt.Println("C'est le matin !")
	case hour > 12 && hour < 18:
		fmt.Println("C'est l'après midi")
	default:
		fmt.Println("On est le soir")
}
```

While

> Attention !
> Le `while` n'existe pas en Go ! Haha !

For

Le `for` en Go est très évolué et permet de remplacer l'utilisation du `while`.

```
sum := 0
for i := 0; i < 10; i++ {
	sum += i
}
fmt.Println(sum)
```

Remplacement du while

```
n := 1
for n < 5 {
    n *= 2
}
fmt.Println(n)
```

>

> Les instructions `continue` et `break` sont aussi utilisables en Go

Boucle infinie

```
for {
    fmt.Println("Boucle infinie")
}
```

Cours

Tableaux

Tableaux à taille fixe

Définition

> Simplement Un tableau à taille fixe est une séquence d'éléments d'une taille définie

- Tout est alloué d'un seul bloc ➡ les cases sont contiguës en mémoire
- Le premier index démarre à 0.
- La taille est définitive, pour agrandir, il faut allouer un autre tableau ou utiliser une autre technique.
- Le contenu sera toujours initialisé à `0, "", ...`.

Syntaxe

```
var nom[taille]type
```

```
var tab[5]int
t[3] = 12
```

Déclaration rapide

```
odds := [4]int{1, 3, 5, 7}
pair := [4]int{2, 4} // [2, 4, 0, 0]
```

Affichage

```
var names [3]string
names[0] = "Bob"
```

```
names[2] = "Alice"


fmt.Printf("name[2]=%v\n", names[2])
fmt.Printf("names=%v (len=%d)\n", names, len(names))
```

Tableaux dynamiques (Slice)

Définition

Tableau de taille dynamique

- Slice ⇒ Tranche □□
- Un Slice représente une tranche d'un tableau.
- Un Slice est une "vue" sur le tableau sous-jacent
- Modifier le slice ➡ modifier le tableau

Syntaxe

```
s := make([]type, taille, capacité)
```

- **Taille** : nombre d'éléments du slice
- **Capacité** (facultatif) : nombre d'éléments du tableau

```
s := make([]int, 3)
s[0] = -3
len(s) // 3
cap(s) // 3
```

Réallocation

```
s := make([]int, 3)
s = append(s, 12)
len(s) // 4
cap(s) // 6
```

Explication :

- Si on dépasse la taille du tableau
- Un nouveau tableau est alloué, de capacité doublée

Sous-tableaux

```
letters := []string{"g", "o", "l", "a", "n", "g"}
fmt.Printf("%v \n", letters)

// subslicing
sub1 := letters[:2]
sub2 := letters[2:]
fmt.Printf("%v\n", sub1) // ?
fmt.Printf("%v\n", sub2) // ?
```

Référence des sous tableaux

Que se passe-t-il si on fait ça ?

```
letters := []string{"g", "o", "l", "a", "n", "g"}
sub1 := letters[:2]
sub2 := letters[2:]

sub2[0] = "UP"
fmt.Printf("%v\n", sub2) // ?
fmt.Printf("%v\n", letters) // ?
```

Et là ?

```
letters := []string{"g", "o", "l", "a", "n", "g"}
sub2 := letters[2:]

subCopy := make([]string, len(sub2))
copy(subCopy, sub1)
subCopy [0] = "UP"
fmt.Printf("%v\n", subCopy) // ?
fmt.Printf("%v\n", letters) // ?
```

Cours

Les fonctions

```
func printInfoNoParam() {
	fmt.Printf("Name=%s, age=%d, email=%s\n", "Bob", 10, "bob@golang.org")
}

func printInfoParams(name string, age int, email string) {
	fmt.Printf("Name=%s, age=%d, email=%s\n", name, age, email)
}

func avg(x, y float64) float64 {
	return (x + y) / 2
}

func sumNamedReturn(x, y, z int) (sum int) {
	sum = x + y + z
	return // c pas bo hein ...
}

func main() {
	printInfoNoParam()
	printInfoParams("Noé", 15, "noe@flex.org")

	result := avg(16.3, 25.0)
	fmt.Printf("Average result=%f\n", result)

	sum := sumNamedReturn(10, 25, 7)
	fmt.Printf("Sum result=%d\n", sum)
}
```

Multiples return

```
func ToLowerStr(name string) (string, int) {
	return strings.ToLower(name), len(name)
```

```
}

func main() {
	lowerName, len := ToLowerStr("NOE") // on a le droit mais c'est pas beau non plus
	fmt.Printf("%s (len=%d)\n", lowerName, len)

	_, len = ToLowerStr("Paul ABIB, oui le seul le vrai l'unique")
	fmt.Printf("bob len=%d\n", len)
}
```

Cours

Range

> C'est la continuité du `for`, il permet d'itérer sur une collection de donnée

Syntaxe

```
for <index>, <value> := <dataset> {
	//code
}
```

Exemple

```
names := []string{"Bob", "Alice", "Bobette", "John"}
	for i, n := range names {
		fmt.Printf("Username=%s (index=%d)\n", n, i)
	}

	// range on string
	// Omit index / value
	for _, c := range "golang" {
		fmt.Printf("%v\n", string(c))
	}
```

Cours

Gestion d'erreurs

Gestion d'erreurs dans les langages

Il y a plusieurs stratégies possibles :

- Code d'Erreurs
- Exceptions
- Pattern Matching
- ...

Go et le retour multiple

En Go, nous allons exploiter le retour multiple des fonctions pour gérer nos erreurs

Exemple classique

```
func MyFunc() (int, error) {
	// code
	return 1
}

func main() {
	v, err := MyFunc()

	if err != nil {
		fmt.Printf("Error in MyFunc: %v", err)
	}
}
```

> nil = NULL

Gestion d'erreur standard en Go

En Go, on peut retrouver souvent des codes qui auront cette forme-là pour gérer les erreurs

```
v1, err := MyFunc1()
if err != nil {
	return err
}

v2, err := MyFunc2()
	if err != nil {
return err
}

v3, err := MyFunc3()
	if err != nil {
return err
}

v4, err := MyFunc4()
	if err != nil {
return err
}
```

C'est un peu répétitif... Mais efficace ! Cela apporte une lecture du code progressivement.

Early return

En Go, on va favoriser les tests et retour d'erreurs en tout début de fonction, pour faire un retour d'erreur le plus rapidement possible, permettre à notre code qui suit de grandir plus facilement.

Code non-early return

```
func MyFunc(condition bool) (int, err) {
	if (condition) {
		if (!condition2) {
			return 0, errors.New("Error 2!")
```

```
    }
    // code
    return 42, nil
  }
  return 0, errors.New("Error!")
}
```

Code early-return

```
func MyFunc(condition bool) (int, err) {
  if (!condition) {
    return 0, errors.New("Error!")
  }
  if (!condition2) {
    return 0, errors.New("Error 2!")
  }
  // code
  return 42, nil
}
```

Cours

Fichiers

Pour manipuler un fichier en Go, il existe plusieurs librairies permettant différentes actions.

io/ioutil

C'est sans doute l'approche la plus simple pour manipuler un fichier.

Elle permet de directement lire un répertoire ou le contenu d'un fichier, et même d'écrire dedans.

Lecture fichier

```
func readFile(filename string) (error) {
	dat, err:= ioutil.ReadFile(filename)
	if err != nil {
		return "", err
	}

	if len(dat) == 0 {
		// return "", errors.New("Empty content")
		return "", fmt.Errorf("Empty content (filename=%v)", filename)
	}

	fmt.Printf("%s\n", dat)

	return nil
}
```

Écriture fichier

```
func writeFile(filename, content string) error {
	err:= ioutil.WriteFile(filename, []byte(content), 0644)
```

```
	if err != nil {
		return err
	}
	return nil
}
```

Lecture répertoire

```
func readDir() error {
	files, err := ioutil.ReadDir(".")
	if err != nil {
		return err
	}

	for _, file := range files {
		fmt.Println(file.Name())
	}
	return nil
}
```

Inconvénient

On fait de la lecture / écriture direct, aucun buffer.

Peut poser problème en cas de traitement de gros fichiers....

On peut potentiellement écraser un fichier existant

os + bufio

`Bufio` implémente des manipulations de flux avec des buffers, un peu plus verbeux, mais beaucoup plus intéressant !

Lecture

```
func readFile(filename string) error {
	srcFile, errSrc := os.Open(src)
	if errSrc != nil {
		return errSrc
	}
	
	lineIdx := 1
	scanner := bufio.NewScanner(srcFile)

	for ; scanner.Scan(); lineIdx++ {
		fmt.Println("Line", lineIdx, ":", scanner.Text())
	}

	srcFile.Close()
	return nil
}
```

Écriture

```
func writeFile(filename string, lines []string) error {
	dstFile, errDst := os.Create(dst)
	if errDst != nil {
		return errDst
	}

	writer := bufio.NewWriter(dstFile)

	for _, line range lines {
		_, errWrt := fmt.Fprintln(writer, line)
		if errWrt != nil {
			return errWrt
		}
	}

	writer.Flush()
	dstFile.Close()
	return nil
}
```

Cours

Defer

Repousser l'exécution d'une instruction

Cas d'utilisation

Dans l'exemple si dessous

```
func main() {
	f := os.OpenFile("foo.txt")
	if condition1 {
		return // Oops...! pas de close ici!
	}
	// code
	f.Close()
}
```

Le `f.close()` peut être très éloigné dans le code du `OpenFile`.
Or, quand on ouvre un fichier, on va surement le fermer ensuite, mais pas tout de suite...

On peut même potentiellement arrêter notre fonction avant de fermer le fichier sans faire attention !

Il paraît donc plus "jolie" de mettre le code d'ouverture du fichier et celui de fermeture côte à côte, car il traite le même sujet.

Solution

```
func main() {
	f := os.OpenFile("foo.txt")
	defer f.Close() // exécuté quand on sort de main()
```

```
	if condition1 {
		return
	}
}
```

> `defer` est rattaché à la fonction qui l'invoque

Ordre d'exécution

Les instructions mises en `defer` fonctionnerons comme une pile **LIFO** (Last In First Out).

Cours

Kata Find and Replace

Énoncé

Programme qui trouve et remplace un mot par un autre dans un fichier.

Exemple

*Remplacer le mot **Go** par **Python***

Source: wikigo.txt	Résultat
Go was conceived in 2007 to improve programming productivity at Google	Python was conceived in 2007 to improve programming productivity at Pythonogle

Features

- Affiche le résumé du traitement en console
  - Nombre d'occurrences du mot
  - Numéros de lignes modifiés
- Écrire le texte modifié dans un nouveau fichier pour préserver l'original
- Bonus : prendre aussi en compte que le mot peut avoir une majuscule (ou pas !)

Exemple de résumé

```
$ go run main.go
== Summary ==
Number of occurrences of Go: 10
Number of lines: 7
Lines: [ 1 - 8 - 15 - 17 - 19 - 23 - 28 ]
== End of Summary ==
```

Prototypes

```
func ProcessLine(line, oldWord, newWord string) (found bool, result string, occurrences int)
```

- `line` : ligne à traiter
- `oldWord` / `newWord` : ancien mot / nouveau mot
- `found` : vrai si au moins une occurrence trouvée
- `result` : résultat après traitement
- `occurrences` : nombre d'occurrences de `oldWord` dans la ligne

```
func FindReplaceFile(src string, dst string, oldWord string, newWord string) (occurrences int, lines []int, err error)
```

- `src` : nom du fichier source
- `oldWord` / `newWord` : ancien mot / nouveau mot
- `occurrences` : nombre d'occurrences de `oldWord`
- `lines` : tableau des numéros de lignes ou `oldWord` a été trouvé
- `err` : erreur générée par la fonction

> Astuce
> `FindReplaceFile` n'arrive pas à ouvrir le fichier, il faut renvoyer une erreur

Outils

Bufio

```
scanner := bufio.NewScanner(srcFile)
for scanner.Scan() {
t := scanner.Text()
}
```

```
writer := bufio.NewWriter(dstFile)
defer writer.Flush()
fmt.Fprintln(writer, "Texte d'une ligne")
```

Strings

```
c := strings.Contains("go ruby java", "go") // c == true
cnt := strings.Count("go go go", "go") // cnt == 3
res := strings.Replace("old go", "go", "python", -1) // res == "old python"
```

Solution

```
package main

import (
	"bufio"
	"fmt"
	"os"
	"strings"
)

func ProcessLine(line, oldWord, newWord string) (found bool, result string, occurrences int) {
	oldWordLower := strings.ToLower(oldWord)
	newWordLower := strings.ToLower(newWord)
	result = line
	if strings.Contains(line, oldWord) || strings.Contains(line, oldWordLower) {
		found = true
		occurrences += strings.Count(line, oldWord)
		occurrences += strings.Count(line, oldWordLower)
		result = strings.ReplaceAll(line, oldWord, newWord)
		result = strings.ReplaceAll(result, oldWordLower, newWordLower)
	}

	return found, result, occurrences
}

func FindReplaceFile(src string, dst string, oldWord string, newWord string) (occurrences int,
lines []int, err error) {
	// open src file
	srcFile, errSrc := os.Open(src)
	if errSrc != nil {
		return 0, lines, errSrc
	}
	defer srcFile.Close()

	// open dst file
	dstFile, errDst := os.Create(dst)
	if errDst != nil {
```

```
    return 0, nil, errDst
  }
  defer dstFile.Close()

  oldWord = oldWord
  newWord = newWord
  lineIdx := 1
  scanner := bufio.NewScanner(srcFile)
  writer := bufio.NewWriter(dstFile)
  defer writer.Flush()

  for scanner.Scan() {
    found, res, o := ProcessLine(scanner.Text(), oldWord, newWord)
    if found {
      occurrences += o
      lines = append(lines, lineIdx)
    }

    _, errWrt := fmt.Fprintln(writer, res)
    if errWrt != nil {
      return occurrences, lines, errWrt
    }

    lineIdx++
  }

  return occurrences, lines, nil
}

func main() {
  oldWord := "Go"
  newWord := "Python"
  occurrences, lines, err := FindReplaceFile("wikigo.txt", "wikipython.txt", oldWord, newWord)
  if err != nil {
    fmt.Printf("Error while executing find replace: %v\n", err)
    return
  }

  fmt.Println("== Summary ==")
  defer fmt.Println("== End of Summary ==")
```

```
	fmt.Printf("Number of occurrences of %s: %d\n", oldWord, occurrences)

	fmt.Printf("Number of lines: %d\nLines: [ ", len(lines))
	linesCount := len(lines)
	for i, l := range lines {
		fmt.Printf("%d", l)
		if i < linesCount-1 {
			fmt.Printf(" - ")
		}
	}
	fmt.Println(" ]")

}
```

Cours

Structures & Pointeurs

Définition

> Simplement
> Type personnalisé représentant une collection de champs

Syntaxe

```
type <NomStruct> struct {
	var1 int
	var2 string
	var3 float64
}
```

Exemple

```
type User struct {
	Name string
	Email string
	Age int
}
```

Déclaration

Il y a 3 types de déclaration possible

```
type Person struct {
	Name    string
	Age     int
```

```
}

func main() {
  // 1
	var p1 Person
	p1.Name = "Bob"
	p1.Addr.city = "Lyon"
	
	// 2
	p2 := Person{"Paul","Abibi"}

	// 3
	p3 := Person{Name: "Swann"}
}
```

Règles

- Une structure ne peut contenir que des variables
- La règle de visibilité de package s'applique pour :
  - la structure elle-même
  - les variables de la structure

Exercice player

- Définir 2 structure :
  - Avatar
    - Url
  - Player
    - Name
    - Age
    - Avatar
    - password
- Le mot de passe doit avoir un scope privé.
- Créer une fonction de création d’un player, qui ne prend que son nom en argument et initialise la structure avec ce dernier, ainsi qu’un mot de passe par défaut.

Solution

```
type Avatar struct {
	Url string
```

```
}

type Player struct {
	Name     string
	Age      int
	Avatar   Avatar
	password string
}

func New(name string) Player {
	return Player{
		Name:     name,
		password: "defaultpassword",
	}
}
```

Embedded struct

```
type Avatar struct {
	Url string
}

type Player struct {
	Name     string
	Avatar   Avatar
}
```

Un Player a un Avatar

Parfois, on veut exprimer un autre type de relation → Un XXX est un YYY

Dans d'autres langages, cette relation est exprimée par l'héritage

Go préfère la composition avec l'embedded struct :

```
type Avatar struct {
	Url string
}
```

```
type Player struct {
	Name     string
	Avatar // Pas de nom de variables
}

var p Player
p.Url = "https://photodemoi.jpg"
```

Dans ce code, Avatar est embarqué dans le type Player

Player est un Avatar

Receiver function

Grâce à ce fonctionnent, on peut enfin reproduire quasiment à l'identique le comportement d'un objet tel qu'en Java par exemple.

```
type User struct {
	Name string
}
func (u User) SayHello() {
	fmt.Printf("Hello %v!\n", u.Name)
}
```

- **u** est un value receiver pour la méthode SayHello()
- Les champs de **u** sont accessibles pour la fonction
- C'est bien beau ça, mais à quoi ça nous sert ?

```
func main() {
    u := User{"Paul"}
    u.SayHello()
}
```

Lorsqu'on utilise cette technique, notre struct User passé en argument est en réalité "copiée" pour la méthode.

Conséquence : Une méthode avec un value receiver ne peut pas modifier la structure originale. Cela peut permettre de favoriser l'immutabilité en renvoyant une nouvelle instance de notre structure avec les propriétés mise à jour !

Exercice rectangle

- Définir une structure rectangle qui contient
  - Longueur
  - Largeur
- Créer 3 receiver functions pour cette structure :
  - `Area()` : renvoie l'air du rectangle
  - `String()` : Affiche les informations de la structure bien formatées
  - `DoubleSize()` : Renvoie une nouvelle structure du rectangle avec sa taille doublée

> Astuce
> Définir la receiver function `String()` d'une structure viendra surcharger l'affichage par défaut de cette dernière !

Solution

```
package main

import (
	"fmt"
)

type Rect struct {
	Width, Height int
}

func (r Rect) Area() int {
	return r.Width * r.Height
}

func (r Rect) String() string {
	return fmt.Sprintf("Rect ==> width=%v, height=%v", r.Width, r.Height)
}

func (r Rect) DoubleSize() Rect {
	r.Width *= 2
	r.Height *= 2
	return r
}
```

```
func main() {
	r := Rect{2, 4}
	fmt.Printf("Rect area=%v\n", r.Area())
	fmt.Println(r)

	r2 := r.DoubleSize()
	fmt.Println("r", r)
	fmt.Println("r2", r2)
}
```

Pointeurs

En Go, lorsque qu'on passe un paramètre à une fonction, on passe en réalité une copie de cette dernière,

Les pointeurs en Go fonctionnent presque comme en C, à l'exception que nous n'avons pas à gérer l'allocation et la libération de la mémoire.

```
x := -42
s := "Bob"
p := &x // Création d'un pointer vers la variable x
i := *p // Déréférencement de p pour récupérer la valeur de x
```

Manipulations

```
func UpdateVal(val string) {
	val = "value"
}

func UpdatePtr(ptr *string) {
	*ptr = "pointer"
}

func main() {
	i := 1
	var p *int = &i
```

```
	fmt.Printf("i=%v\n", i)
	fmt.Printf("p=%v\n", p)
	fmt.Printf("*p=%v\n", *p)
	fmt.Println("---------------")

	s := "Paul"
	sPtr := &s
	s2 := *sPtr
	fmt.Println("String pointer")
	fmt.Printf("*s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("---------------")

	*sPtr = "Clément"
	fmt.Println("Dereference and update")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("---------------")

	UpdateVal(s)
	fmt.Println("Func UpdateVal()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("---------------")

	UpdatePtr(&s)
	UpdatePtr(sPtr)
	fmt.Println("Func UpdatePtr()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("---------------")
}
```

Pointer Receiver

Comme dit plus tôt, les paramètres de fonctions sont des copies des objets originaux.

Ça vaut aussi pour les fonctions `value receiver` sur les structures. Elles ne peuvent donc que faire de la lecture simple, et ne peuvent pas modifier la structure d'origine.

Grâce aux pointeurs, on peut régler ce problème et le couplant à des `receiver functions`.

```
type Post struct {
	Title     string
	Text      string
	published bool
}

func (p Post) Headline() string {
	return fmt.Sprintf("%v - %v", p.Title, p.Text[:50])
}

func (p *Post) Publish() {
	p.published = true
}

func (p *Post) Unpublish() {
	p.published = true
}

func main() {
	p := Post{
		Title: "Go release",
		Text: "Go is a programming language...",
	}

	fmt.Println(p.Headline())

	fmt.Printf("Post published? %v\n", p.Published())
	p.Publish()
	fmt.Printf("Post published? %v\n", p.Published())
}
```

Enfin, si on souhaite créer une structure directement sous la forme d'un pointeur, on peut faire autrement que :

```
p := Post{
    Title: "Go release",
    Text: "Go is a programming language...",
}
pointer := &p
```

Comme ceci :

```
pythonPost := &Post{
    Title: "Python Intro",
    Text: "Python is an interpreted high-level programming language",
}
```

`pythonPost` est un pointeur.

Cours

Maps

Définition

Structure associant des clés à des valeurs

On peut mettre en clé tout ce qui est comparable (on peut mettre une structure comme clé)

Syntaxe

La syntaxe “longue” de déclaration d’une map est la suivante :

```
var m map[KeyType]ValueType
---------------------------
var m map[string]int = make(map[string]int)
```

Grâce à l’inférence des types à la déclaration, on peut encore simplifier cette syntaxe en :

```
m2 := make(map[string]int)
```

Opérations

Pour ces exemples, nous avons une map qui a pour clé une chaine de caractère et pour valeur un entier.

```
myMap := make(map[string]int)
```

On peut récupérer la taille de map en utilisant une fonction que nous connaissons depuis les slices `len()`

```
fmt.Printf("Map size %v\n", len(myMap))
```

Assignation

L’assignation et très simple, trèèèèès simple !

```
myMap["hello"] = 5
myMap["goodbye"] = 10
```

Récupération

Pour récupérer la valeur, comme pour l'assignation, il suffit de faire comme avec un tableau

```
fmt.Printf("key=hello, value=%v\n", myMap["hello"])
```

Présence d’une clé

Pour tester la présence, on utilise le retour multiple caché dans la récupération d’une valeur

```
j, isPresent := myMap["helo"]
fmt.Printf("j=%v, isPresent =%v\n", j, isPresent )
```

`isPresent` est un type booléen et est égale à `false` si la clé n’existe pas.

Dans le cas où la clé n’existe pas, la valeur sera celle par défaut du type (0 pour un entier, chaine vide pour une chaine de caractères, ...)

Si on souhaite mettre ce test dans une condition, on peut faire de cette manière :

```
if _, present = myMap["hel"]; present {
// ... code
}
```

Supprimer une clé / valeur

On utilise la fonction `delete`

```
delete(myMap, "hello")
```

Assignation rapide et parcours

On peut assigner des valeurs dans notre map dès la déclaration comme avec les tableaux

```
myMap := map[string]int{
"Noé": 10,
"Paul": 15,
"Swann": 18,
"Nathanael" : 0
}
```

Pour parcourir une map, on peut utiliser `range`

```
for name, idk := range myMap{
fmt.Printf("name=%v, idk=%v\n", name, idk)
}

// Only keys
for name := range m {
fmtfmt.Printf("name=%v\n", name)
}
```

Map & Struct

Pour illustrer la mise en place d’une map constitué de structures, on va utiliser les structures suivantes

```
type User struct {
name string
}

type Key struct {
ID   int
Name string
}
```

On peut créer une map de cette manière :

```
myMap := make(map[Key]User)

myMap[Key{1,"ceo"}] = User{"Swann"}

fmt.Printf("%v"), myMap[Key{1,"ceo"}])
```

> Lorsqu'on récupère une structure depuis une map, on récupère en réalité une copie de cette dernière. Pour palier à ça on peut transformer notre map en une map de pointeurs

```
myMap := make(map[Key]*User)

myMap[Key{1,"ceo"}] = &User{"Swann"}

fmt.Printf("%v"), *myMap[Key{1,"ceo"}])
```

Gin Framework

Gin est un framework web écrit en Go (Golang).

Introduction à Gin

Repo github des exercices

Présentation de Gin

Gin est un framework web HTTP écrit en Go.

Il dispose d'une API de type Martini, mais avec des performances jusqu'à 40 fois plus rapides que Martini. Si vous avez besoin de performances époustouflantes, procurez-vous du Gin (le framework hein !).

Gin simplifie de nombreuses tâches de codage associées à la création d'applications Web, y compris les services Web.

Fonctionnalités

Documentation Gin

- Rapide
- Prise en charge des middlewares
  - Exemple : Logger, Authorization, GZIP ...
- Pas de crash
  - Possède un catcheur □□♂ interne pour intercepter les erreurs et empêcher l’arrêt de notre API
- JSON Validation
- Gestion d’erreur
  - On peut gérer manuellement les erreurs interceptées

Gin VS Node

Gin est INCROYABLEMENT rapide comparé à des concurrents de tous les jours.

Le fait qu’il soit codé en Go permet d’obtenir un binaire complet pesant ~10Mo et contenant notre serveur API au complet. Comparé à Node qui doit inclure toutes ses librairies, c’est beaucoup moins !

Untitled

Untitled

Docker

On peut créer des images docker de notre API gin ULTRA légères, puisque le binaire est standalone, une image alpine suffit □□

```
ARG GO_VERSION=1.18

FROM golang:${GO_VERSION}-alpine AS builder

RUN apk update && apk add --no-cache alpine-sdk git

WORKDIR /api

COPY go.mod .
COPY go.sum .
RUN go mod download

COPY . .
RUN go build -o ./app ./main.go

FROM alpine:latest

RUN apk update && apk add --no-cache ca-certificates

WORKDIR /api
COPY --from=builder /api/app .

EXPOSE 8080

ENTRYPOINT ["./app"]
```

> On peut faire plus simple, mais je vous montre une version très optimisée d’un Dockerfile pour faire tourner une app go dans un environnement ultra léger.

Installation

> Dans le cadre de cette explication, j'utiliserais l'IDE Goland, donc il se peut que certaines choses soient simplifiés par l'IDE, et d'autres que je doive faire spécifiquement par rapport à cet IDE

Nouveau projet

Tout d'abord, nous allons créer un nouveau projet Go.

Untitled

> Dans l'environnement, j'ai spécifié `GOPROXY=direct`
> C'est très important, car cela va nous permettre d'inclure des librairies externes (Gin)

Vérifier dans vos paramètres Go / Go Modules que l'option **Enable Go Modules integration** est bien coché et ressemble à ça :

Untitled

Ajout des dépendances

Pour ajouter les dépendances nécessaires à Gin, nous devons ouvrir un terminal à la racine du projet (que ne dois contenir pour le moment que le fichier `go.mod`) et taper la commande suivante.

```
go get -u github.com/gin-gonic/gin
```

Cela va télécharger et indiquer dans notre fichier `go.mod` les dépendances nécessaires au fonctionnement de **Gin** :

Untitled

Création d’un serveur basique

Pour tester que tout va bien, nous allons créer le fichier `main.go` à la racine du projet et le remplir ainsi :

```
package main

import "github.com/gin-gonic/gin"

func main() {
	r := gin.Default()

	r.GET("/", func(c *gin.Context) {
		c.JSON(200, gin.H{
			"message": "hello world",
		})
	})

	r.Run(":8080")
}
```

Testons ce petit code rapidement avec la commande :

```
go run main.go
```

Untitled

Et voilà ! Vous avez votre premier serveur Gin qui est en marche !

Rendons-nous avec notre navigateur sur l’adresse localhost:8080 pour admirer la superbe réponse

Untitled

Untitled

Live reload

Recompiler notre code à chaque fois que l'on change notre code, arrêter le serveur et le relancer...

Tout ça est long et fastidieux ! Surtout pendant le développement !

En nodeJS certains se souviendrons de nodemon qui permettait de surveiller les changements dans nos fichiers, et les recompiler à la volée.

En Go il existe différents outils permettant de faire cela, nous allons utiliser Air

Air installation

Pour installer Air en tant qu'exécutable reconnu par notre machine, il suffit de faire la commande suivante :

```
go install github.com/cosmtrek/air@latest
```

Ensuite, nous allons "pimper" un peu la configuration de cet outil pour avoir un peu de couleur □□.

Pour ce faire, nous allons créer le fichier `.air.conf` à la racine de notre projet et le remplir ainsi :

```
# .air.conf
# Config file for [Air](https://github.com/cosmtrek/air) in TOML format

# Working directory
# . or absolute path, please note that the directories following must be under root.
root = "."
tmp_dir = "tmp"

[build]
# Just plain old shell command. You could use `make` as well.
cmd = "go build -o ./tmp/main ." # replace by main.exe if on windows !
# Binary file yields from `cmd`.
bin = "tmp/main" # replace by main.exe if on windows !
# Customize binary.
```

```
# Watch these filename extensions.
include_ext = ["go", "tpl", "tmpl", "html"]
# Ignore these filename extensions or directories.
exclude_dir = ["assets", "tmp", "vendor", "frontend/node_modules"]
# Watch these directories if you specified.
include_dir = []
# Exclude files.
exclude_file = []
# It's not necessary to trigger build each time file changes if it's too frequent.
delay = 1000 # ms
# Stop to run old binary when build errors occur.
stop_on_error = true
# This log file places in your tmp_dir.
log = "air_errors.log"

[log]
# Show log time
time = false

[color]
# Customize each part's color. If no color found, use the raw app log.
main = "magenta"
watcher = "cyan"
build = "yellow"
runner = "green"

[misc]
# Delete tmp directory on exit
clean_on_exit = true
```

> Attention à la ligne 11 et 13 !

Désormais, il suffit de lancer la commande `air` à la racine du projet pour le lancer et surveillez les changements de code :

Untitled

Au moindre changement, on pourra voir que le script le détecte et recompile aussitôt

Untitled

Restful API Server

Simple Server

GitHub repo

Pour utiliser **Gin**, il suffit d'importer `github.com/gin-gonic/gin` au niveau de son fichier main et de créer une variable qui va contenir notre fameux routeur.

```
package main

import "github.com/gin-gonic/gin"

func main() {
	router := gin.Default()
}
```

Il faut ensuite lui définir des routes sur lesquels il va écouter. Dans notre exemple, nous ferons 2 GET qui renvoient un JSON facilement grâce à la librairie **Gin.**

```
package main

import "github.com/gin-gonic/gin"

func main() {
	router := gin.Default()

	router.GET("/", func(c *gin.Context) {
		c.JSON(200, gin.H{
			"message": "Hello World!",
		})
	})
	router.GET("/ping", func(c *gin.Context) {
		c.IndentedJSON(200, gin.H{
			"message": "pong",
```

```
□□})
□})
}
```

Enfin, il faut lui dire de se lancer et d'écouter sur un port spécifique grâce à une dernière ligne.

```
package main

import "github.com/gin-gonic/gin"

func main() {
□router := gin.Default()

□router.GET("/", func(c *gin.Context) {
□□c.JSON(200, gin.H{
□□□"message": "Hello World!",
□□})
□})
□router.GET("/ping", func(c *gin.Context) {
□□c.IndentedJSON(200, gin.H{
□□□"message": "pong",
□□})
□})

□router.Run(":9090")
}
```

Dans ce code, nous :

- Initialisons un routeur Gin en utilisant Default.
- Utilisons le GET, pour associer la méthode HTTP `GET` et le chemin `/, /ping` à une fonction contenant le contexte de la requête
- Utilisons `c.JSON` ou bien `c.IndentedJSON` permettent de simplement convertir une structure (en l'occurrence `gin.H` qui permet d'en créer une à la volée)

Il suffit alors de lancer notre magnifique app avec la commande `go run main.go` et aller tester nos routes.

Untitled

Untitled

Aussi simple que ça.

Simple music server

GitHub repo

Pour complexifier un peu plus les choses, on va refaire la même chose, mais avec quelques structures et un peu de découpage. L'objectif étant de servir une API de gestion d'une liste d'album de musique (Très originale oui).

On va essayer de faire du pseudo MVC et l'architecture de notre application sera la suivante :

```
controllers/
|  controller.go
|  command.go
|  query.go
data/
|  albums.go
models/
|  album.go
main.go
go.mod
go.sum
```

Models

Nous allons dans un premier temps définir la structure d'un album. Il faut déclarer dans le fichier `models/album.go` la structure suivante :

```
package models

type Album struct {
ID     string  `json:"id"`
Title  string  `json:"title"`
Artist string  `json:"artist"`
Price  float64 `json:"price"`
}
```

Les balises telles que `json:"artist"` spécifient le nom d'un champ lorsque le contenu de la structure est sérialisé en JSON.

Sans eux, le JSON utiliserait les noms de champs des propriétés, avec la majuscule, ce qui n'est pas très courant (pour rappel, en go, la majuscule, en début de variable ou fonction, permet de définir sa visibilité en dehors de son package) .

Data

On va ensuite déclarer une liste d’albums qui nous serviront de “base de données” pour notre application.

Remplissons dans le fichier `data/albums.go` de cette manière :

```
package data


import "gin-form/simple_music_api/models"


var Albums = []models.Album{
□{
□□ID:     "1",
□□Title:  "Taste of you",
□□Artist: "Rezz",
□□Price:  1.99,
□},
□{
□□ID:     "2",
□□Title:  "Go",
□□Artist: "Google",
□□Price:  9999,
□},
□{
□□ID:     "3",
□□Title:  "C#",
□□Artist: "Microsoft",
□□Price:  -1,
□},
}
```

Controllers

Occupons-nous de la partie controllers désormais !

Dans un premier temps, nous allons écrire nos fonctions servant à récupérer les données uniquement (en mode CQS tu connais).

Le fichier `controllers/query.go` va contenir deux fonctions :

- Une permettant de récupérer la liste des albums
- Une autre permettant de récupérer un seul album par son **ID**

La première partie du fichier ressemblera simplement à ça

```
package controllers

import (
	"gin-form/simple_music_api/data"
	"github.com/gin-gonic/gin"
	"net/http"
)

func getAlbums(c *gin.Context) {
	c.IndentedJSON(http.StatusOK, data.Albums)
}
```

Dans ce code, nous :

- Écrivons une fonction `getAlbums` qui prend un `gin.Context` en paramètre.
  - `gin.Context` est la partie la plus importante de Gin. Il prend en charge la requête, les détailles, la validation et sérialisation JSON, et plus encore.
- Appelons la fonction `c.IdentedJSON` afin de sérialiser notre tableau `data.Albums` en JSON indenté proprement.
- Utilisons une librairie interne à go `net/http` pour récupérer le code HTTP voulu (200). On pourrait écrire directement 200 à la main, mais maintenant vous savez que cette librairie existe

---

La deuxième méthode est un peu plus complexe et permet de récupérer un album parmi ceux existants avec son ID, qui sera passé dans le chemin de la requête (`/album/:id`).

```
func getAlbumByID(c *gin.Context) {
	id := c.Param("id")

	for _, album := range data.Albums {
		if album.ID == id {
```

```
		c.IndentedJSON(http.StatusOK, album)
		return
	}
}
c.IndentedJSON(http.StatusNotFound, gin.H{"error": "Album not found"})
}
```

Nous allons maintenant nous occuper du fichier `controllers/command.go` qui contiendra notre fonction permettant d'ajouter un album à notre liste.

```
package controllers

import (
	"gin-form/simple_music_api/data"
	"gin-form/simple_music_api/models"
	"github.com/gin-gonic/gin"
	"net/http"
)

func addAlbum(c *gin.Context) {
	var newAlbum models.Album

	if err := c.BindJSON(&newAlbum); err != nil {
		c.IndentedJSON(400, gin.H{
			"message": "Invalid JSON",
			"error":   err.Error(),
		})
		return
	}

	data.Albums = append(data.Albums, newAlbum)
	c.IndentedJSON(http.StatusCreated, newAlbum)
}
```

Dans cette fonction nous :

- Déclarons une variable `newAlbum` de type `Album`
- Utilisons la méthode fournit par **Gin** `c.BindJSON` qui va tenter de parser le body de notre requête dans notre structure en se basant sur le format décrit plus haut (les fameux `json:"artist"`).

    - Si, on n’y parvient pas, on renvoie un code erreur avec un message et stoppons l’exécution de la méthode.
- Ajoutons notre nouvel album à notre tableau
- Renvoyons un code de Création et l’album qui vient d’être enregistré

Les fonctions écrites plus hautes sont privées, il va falloir donc faire quelque chose pour qu’elles puissent être utilisées par notre routeur se trouvant dans le fichier `main.go`.

Ça sera le but de notre fichier `controllers/controller.go` qui va s’occuper de faire notre routage :

```
package controllers

import "github.com/gin-gonic/gin"

func SourceControllers(router*gin.Engine) {
router.GET("/albums", getAlbums)
router.GET("/albums/:id", getAlbumByID)
router.POST("/albums", addAlbum)
}
```

Nous pouvons enfin relier tout ça à notre routeur principal dans le fichier `main.go`

```
package main

import (

"gin-form/simple_music_api/controllers"
"github.com/gin-gonic/gin"
)

func main() {
router := gin.Default()

controllers.SourceControllers(router)

router.Run(":8080")
}
```

On peut maintenant aller tester tout ça

Untitled

Untitled

Untitled

Untitled

Untitled

GGWP □□

Static server

GitHub repo

Dans certains cas, on souhaite juste héberger un site statique.

On pourrait se tourner vers apache ou nginx mais ce n'est pas ce que nous recherchons □□

Il est possible assez facilement grâce à **Gin** de rendre accessible notre site statique.

Pour cette démonstration, je possède l'architecture suivante :

```
static/
|  assets/
|  |  ...
|  index.html
|  script.js
|  ...
main.go
go.mod
go.sum
```

Mon site dans le dossier `static` est une application Angular (avec Angular router pour l'exemple haha) compilé en version de production.

Pas besoin d'aller très loin, notre fichier `main.go` ressemblera à ça :

```
package main


import "github.com/gin-gonic/gin"


func main() {
□router := gin.Default()


□router.Static("/", "./static")


□router.Run(":8080")
}
```

Pas besoin de détailler, les fonctions parlent d'elles même.

Si je me rends sur mon site, on voit que tout va BIEN :

Untitled

Je me rends sur une autre page de mon site en lançant un combat, et là aussi tout va BIEN :

Untitled

> SAUF QUE !
> Si je décide de rafraîchir ma page, avec cet URL là, et bien j'obtiens une belle 404...

Untitled

Cela vient du fait que le router va chercher bêtement un fichier au chemin `/fight/charizard/blastoise` dans notre dossier `static` alors qu'il devrait passer ce chemin à notre application Angular, c'est un problème récurent avec les applications web.

Il existe heureusement une solution, il suffit de dire à **Gin** que s'il ne trouve pas le chemin en question dans l'arborescence de dossier, il doit alors interroger l'application Angular, qui se chargera elle-même de renvoyer une erreur 404 si le chemin n'existe effectivement pas.

```
package main

import "github.com/gin-gonic/gin"

func main() {
	router := gin.Default()

	router.Static("/", "./static")

	router.NoRoute(func(c *gin.Context) {
		c.File("./static/index.html")
	})

	router.Run(":8080")
}
```

Et là, si on rafraichit, on retrouve notre beau combat dans notre arène

Reverse proxy

Reverse proxy simple

GitHub repo

Le reverse proxy est quelque chose de majoritairement utilisé aujourd'hui.

Dans beaucoup de cas d'utilisation, on utilise des outils tels que Nginx, Apache, Caddy uniquement pour faire du reverse proxy.

Mais avec **Gin**, on peut coder ça soit même !

Contexte :

- Notre serveur **Gin** écoute en local sur le port 8080
- Mon serveur portainer tourne en local et écoute sur le port 9000
- Je veux qu'en me connectant sur mon serveur **Gin**, ce dernier me fasse un reverse proxy sur mon serveur portainer.

Dans mon fichier `main.go`, nous allons déclarer l'URL de mon reverse proxy ainsi qu'une méthode `proxy` qui sera la méthode utilisée par **Gin**

```
package main


import "github.com/gin-gonic/gin"


const reverseServerAddr = "http://127.0.0.1:9000"


func proxy(c *gin.Context) {


}


func main() {

}
```

Nous dire à notre routeur **Gin**, que TOUTES les requêtes, et ce, peu importe la méthode, doit utiliser notre fameuse méthode `func proxy(c *gin.Context)`.

```
func main() {
	router := gin.Default()

	router.Any("/*any", proxy)

	router.Run(":8080")
}
```

- `router.Any` signifie que quelle que soit la méthode (GET, POST, PUT, ...) utilisé, elle sera pris en charge.
- `/*any` est une expression indiquant à **Gin** que la route peut être n'importe quoi

---

Nous allons maintenant nous attaquer à la méthode `func proxy(c *gin.context)` qui va dans un premier temps parser notre URL de destination (la variable `reverseServerAddr`) avec la librairie `net/url` :

```
func proxy(c *gin.Context) {

	proxy, err := url.Parse(reverseServerAddr)
	if err != nil {
		fmt.Printf("Error parsing reverse proxy address: %s\n", err)
		c.IndentedJSON(http.StatusInternalServerError, gin.H{
			"message": "Error parsing reverse proxy address",
			"error":   err.Error(),
		})
		return
	}

}
```

On gère bien évidement le cas d'erreur où on n'arriverait pas à parser correctement cette URL et on gère le renvoie d'une erreur au client, on arrête également la méthode avec `return`.

La variable `proxy` sera du type `*url.URL`.

Il suffit ensuite d'extraire la requête de notre contexte `c *gin.Context` et modifier son chemin ainsi que son protocole par celui de notre `proxy`.

```
func proxy(c *gin.Context) {
	
	...

	req := c.Request
	req.URL.Scheme = proxy.Scheme
	req.URL.Host = proxy.Host
}
```

Notre requête est prête, nous allons maintenant l'exécuter et récupérer son retour

```
func proxy(c *gin.Context) {
	
	...

	transport := http.DefaultTransport
	resp, err := transport.RoundTrip(req)
	if err != nil {
		fmt.Printf("Error making request: %s\n", err)
		c.IndentedJSON(http.StatusInternalServerError, gin.H{
			"message": "Error making request",
			"error":   err.Error(),
		})
		return
	}

}
```

Là encore, si une erreur survient, on la dirige correctement et on met fin à l'exécution de la méthode.

- `http.DefaultTransport`
- `transport.RoundTrip(req)`

Ce sont des méthodes de la librairie `net/http` et permettent d'exécuter une seule requête web.

Maintenant que la requête a été effectuée et que son retour est récupéré, il faut maintenant la donner à notre réponse et notre reverse proxy sera complet.

```
func proxy(c *gin.Context) {
	
	...
```

```
	for headerKey, headerValues := range resp.Header {
		for _, headerValue := range headerValues {
			c.Header(headerKey, headerValue)
		}
	}
	defer resp.Body.Close()
	bufio.NewReader(resp.Body).WriteTo(c.Writer)
	return
}
```

Ce que nous faisons ici est :

- Nous récupèrerons l'en-tête de notre réponse et les passons à notre contexte (notre vraie réponse).
- Nous passons le body de notre réponse à celui de notre retour aussi.

C’est partit pour tester tout ça !

On lance notre application et on se rend sur notre adresse `localhost:8080`

Untitled

Untitled

Untitled

Load Balancer

Surprise ….