Golang

10 février 2022

Me, my and I

Directeur technique
Zenika Bordeaux

- développement
- formation
- conseil

18 ans d'expérience

Gopher depuis 6 ans

podcast

seb_express

slack

Au programme

- Un peu d’histoire
- Premiers pas
- Advanced
- Liens

Un peu d’histoire

Créé en 2007

Open source en 2009

Release 1.0 en 2012

Inspiré entre autre de C et de Pascal

Robert Griesemer

- JVM

Rob Pike

- UTF-8

Ken Thompson

- langage B
- UTF-8

Les objectifs - la philosophie

👉 Langage fiable et robuste

👉 Langage facile à apprendre

👉 Code rapide à compiler et à exécuter

👉 Les maîtres mots : portabilité, performance et productivité

“Les objectifs du projet GO sont d’éliminer la lenteur et la maladresse du développement logiciel à Google et donc de rendre ce processus plus robuste et plus scalable.”

“La complexité se multiplie.”

“GO est un langage de programmation designé par Google pour aider à résoudre les problèmes de Google et Google a de gros problèmes.”

Rob Pike

Les avantages

- 👉 Langage compilé statiquement
- 👉 Gestion de la mémoire reposant sur un Garbage Collector
- 👉 Cross Platform
- 👉 Mécanisme de concurrence simple et natif (gestion du multi-coeur)
- 👉 Le tooling est fourni avec le langage (linter, test, formatage, LSP, ...)
- 👉 Utilisation de pointeur sans arithmétique de pointeur
- 👉 Retour multiple (tuple)

Applications écrites en GO

- Hugo
- Traefik
- InfluxDB
- Docker
- Kubernetes
- Grafana
- Terraform

- Leboncoin
- Des composants de YouTube

Premiers pas

- La structure d’un programme
- Les types de base
- Les types composés
- Les interfaces & les fonctions
- La gestion des erreurs
- Les tests
- La concurrence

La structure d’un programme

Fichiers avec l’extension go

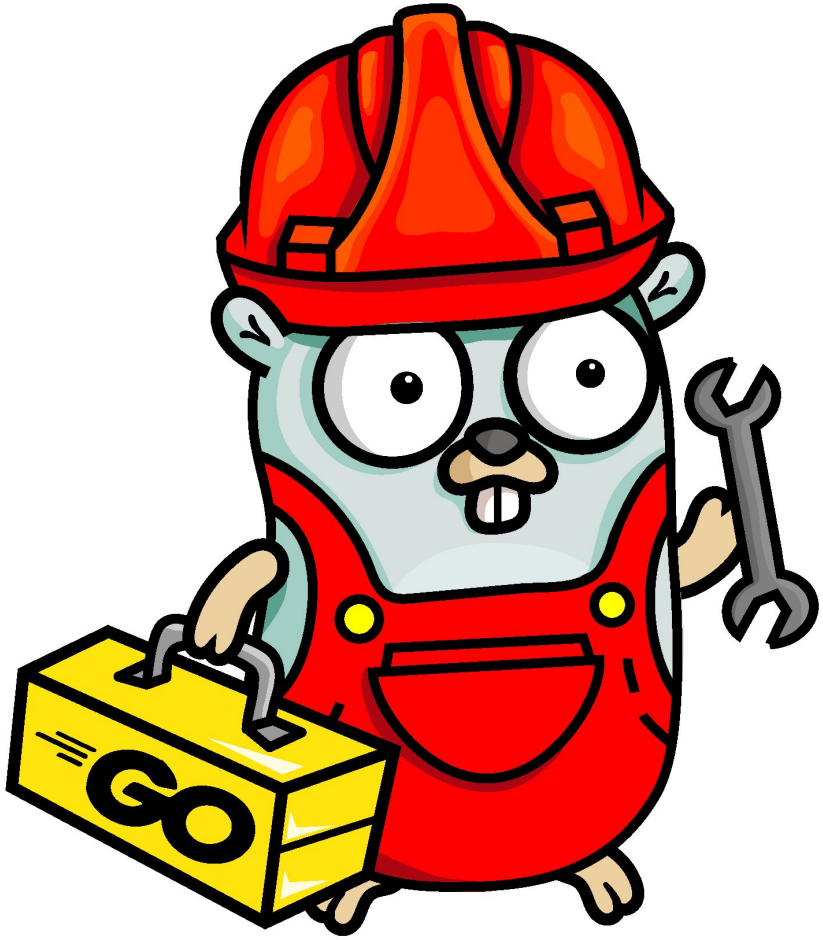
Découpage en package (package main obligatoire)

Structure flat des packages comparativement à Java

2 niveaux de visibilités seulement : privé (commence par une minuscule) et public (commence par une majuscule)

Par convention, les noms de fichiers sont en minuscule

Pas de **bégaiement** dans le nommage.


GO

Les types de base - https://golang.org/pkg/builtin

25 mots clés : break, default, func, interface, select, case, defer, go, map, struct, chan, else, goto, package, switch, const, fallthrough, if, range, type, continue, for, import, return, var

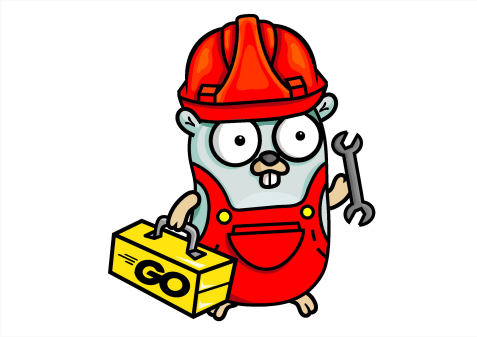
Constants : true, false, iota, nil

Types : int, int8, int16, int32, int64, unit, uint8, uint16, uint32, uint64, uintptr, float32, float64, complex128, complex64, bool, byte, rune, string, error

Fonctions : make, len, cap, new, append, copy, close, delete, complex, real, imag, panic, recover

Déclaration et affectation

- déclaration : **var**
- inférence de type si nécessaire
- affectation : **=**
- déclaration avec affectation : **:=**



Les types composés

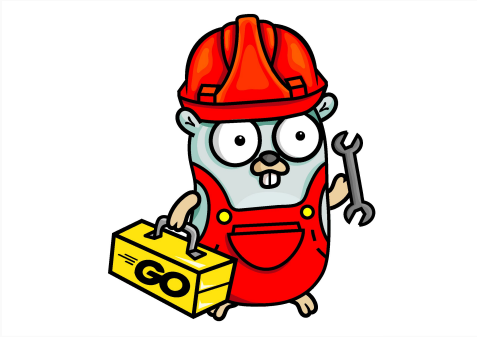
Arrays

Type *primitif*, de **taille fixe**

```
var a [3]int
```

utilisation de la fonction **len** pour la taille

utilisation de **range** pour le parcours



Les types composés

Slices

Permet de gérer des séquences de tailles variable, dynamique

```
var a []int
```

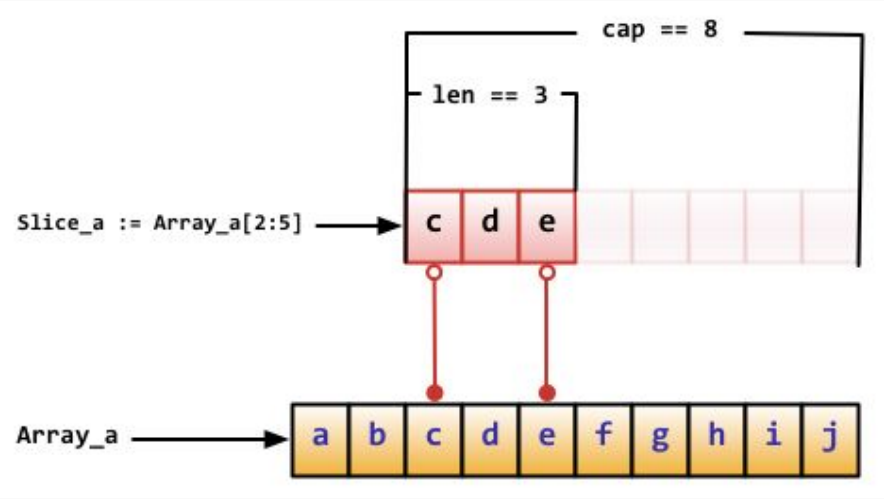
Conceptuellement contient :

- un pointeur sur un array
- un int pour sa longueur
- un int pour sa capacité

Utilisation de la fonction **len** pour sa taille, et **cap** pour sa capacité

Utilisation du mot clé **range** pour le parcours

Les types composés



https://astaxie.gitbooks.io/build-web-application-with-golang/content/en/02.2.html

Utilisation de la fonction **append** pour modifier le slice et **make** pour le déclarer.

```
a := []int{1,2}
a = append(a, 9)
```

```
make([ ]T, len)
make([ ]T, len ,cap)
```

Les types composés

Maps

Une map est une collection non ordonnée

```
ages := map[string]int {“alice”: 21, “bob”: 22}
```

Utilisation possible de **make** pour la déclaration

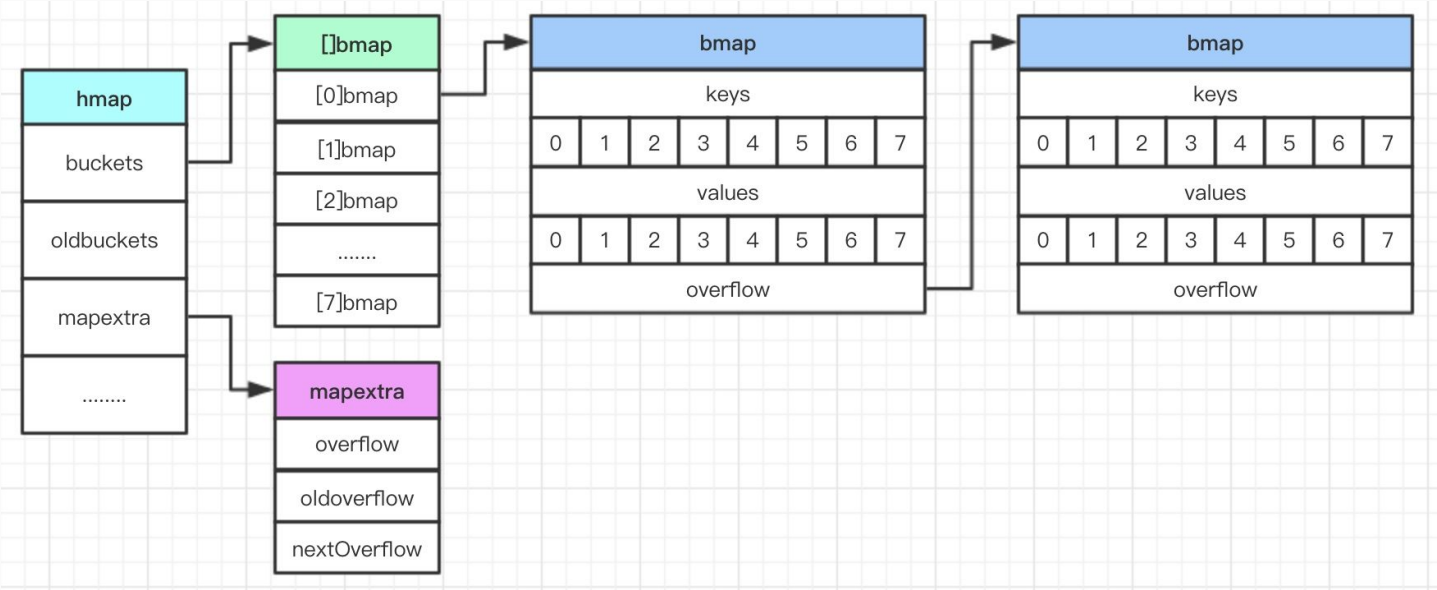
Utilisation de **delete** pour la suppression

Utilisation de **range** pour le parcours

Les types composés

Maps

Une map est une référence sur une hmap.



source : https://www.programmerall.com/article/7309649948/

Les types composés

Structs

Structure composée de champs.

```
type Employee struct {
    ID int
    Name string
    Salary int
}

var dilbert Employee
dilbert = Employee{8, “Dilbert”, 10000}
sebastien := Employee{ID: 12, Name: “Sebastien”}
```

Si tous les champs du struct sont comparable alors le **struct** est comparable via ‘==’

Les fonctions

Func

```
func name(parameters) (returns) {
    body
}
```

```
func hypot(x,y float64) float64 {
    return math.Sqrt(x*x + y*y)
}
```

Les fonctions sont des “first-class citizen”, elles peuvent donc être affectée à des variables ou retournées par des fonctions 😊

Les fonctions

Une fonction peut être rattachée à un **struct** ou ***struct** et ainsi devenir une de ces méthodes.

```
func (t Triangle) area() float64 {
    return (t.base * t.hauteur) / float64(2)
}
```

Une fonction peut retourner plusieurs valeurs dont un objet **Error**.

Il est aussi possible de nommer les variables de retours.

Les fonctions

Les fonctions variadiques sont disponibles en GO.

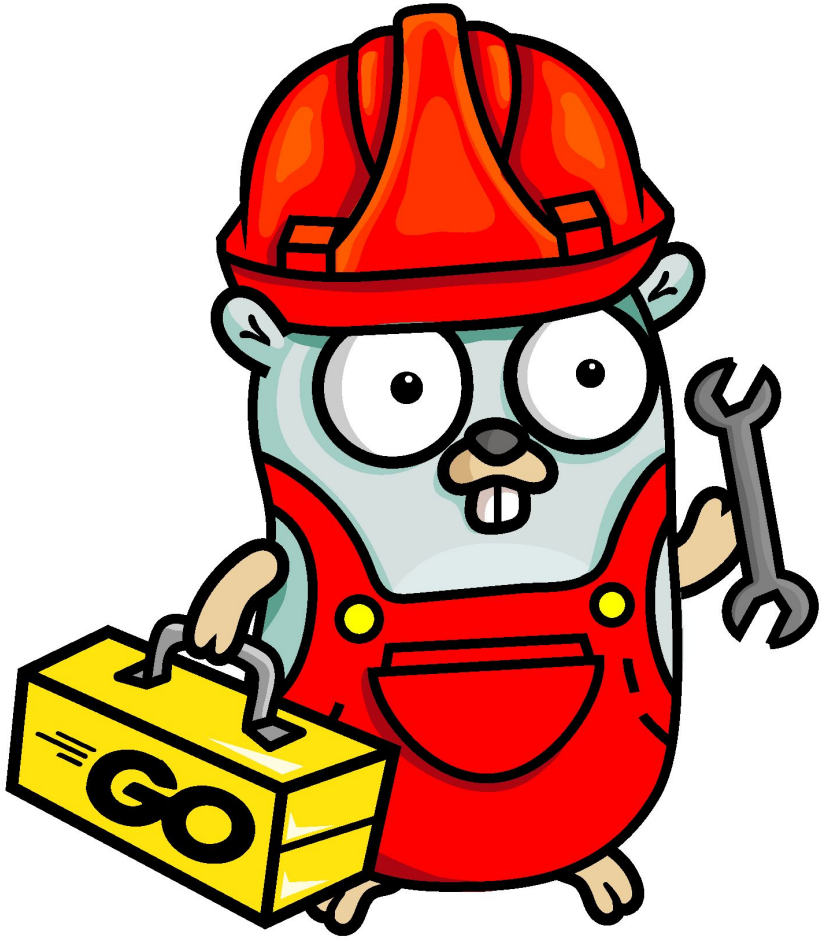
```
func sum(vals ...int) int {
    total := 0
    for _,v := range vals {
        total += v
    }
    return total
}
```

Les fonctions

Il est aussi possible de *déférer* l'appel d'une méthode, pour s'assurer qu'elle sera appelée même si un panic survient.

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Close()
    …
    return nil
}
```

Le **defer** est appelé juste après le return donc si on nomme notre variable de retour, il est possible de la modifier avant qu'elle ne soit modifiée.


GO

Les interfaces

Les interfaces permettent de définir des contrats. En GO les contrats sont généralements très courts (composés d'une seule méthode).

```
package io

type Writer interface {
    Write(p [ ]byte) (n int, err error)
}


package fmt

type Stringer interface {
    String() string
}
```

Les interfaces

En GO, “si tu ressembles à un canard alors tu es un canard”.

C’est ce qu’on appelle le **duck typing** (typage dynamique donc au runtime).

Pour être plus précis, GO repose sur du **structural typing** (typage statique, à la compilation).

Il n’y a donc besoin d’aucun mot clé pour implémenter une interface contrairement à d’autre langage (implements en Java).

Les interfaces

On favorise la composition.

```
type ReadWriter interface {
    Reader
    Writer
}
```

Conceptuellement, la valeur d'un type d'interface est composé d'un type concret et d'un pointeur sur la valeur de ce type.

time.Time
sec: 4564646, loc: "UTC"

La gestion des erreurs

Pas de mécanisme d'exception, les erreurs sont donc à gérer manuellement lors de chaque appel à une méthode qui en prévoit.

On retrouve donc souvent le pattern suivant :

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

La gestion des erreurs

Une **error** est un objet qui répond au contrat d'interface suivant :

```
type error interface {
    Error() string
}
```

Pour créer une erreur, il existe une méthode built-in.

```
errors.New("Not found")
```

La gestion des erreurs

Il y a donc plusieurs façon de gérer les erreurs :

- par type
- par valeur

Depuis Go 1.13, il est possible de **Unwrap** les error afin de remonter la stack des erreurs.

Il existe toutefois des méthodes comme **errors.Is** et **error.As** qui vont permettre de rechercher des types d'erreurs.

👉 https://blog.golang.org/go1.13-errors

La gestion des erreurs

Une bonne pratique est de gérer les comportements des erreurs et non seulement leur type.

👉 https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully

```
type temporary interface {
    Temporary() bool
}

// IsTemporary returns true if err is temporary.
func IsTemporary(err error) bool {
    te, ok := err.(temporary)
    return ok && te.Temporary()
}
```

La gestion des erreurs

Il est possible de traiter des erreurs qui se propagent.

```
panic(“x is nill”)
```

```
if p := recover(); p != nil {
    ...
}
```

Comme son nom l’indique, panic ne doit être utilisé que lorsque l’on considère qu’on ne peut plus continuer le programme.

Les tests

GO vient un avec un tooling pour les tests **go test.**

Les fichiers de test doivent être suffixé par **_test** (file_test.go par exemple) afin que l'outil de test puisse les scanner.

Les fichiers de test peuvent traiter 3 types de fonctions : **tests**, **benchmark** et **example**.

Les tests - test

Une méthode de test doit commencer par **Test**.

```
func TestHelloWorld(t *testing.T) {
    expected := "Hello, World!"
    if observed := HelloWorld(); observed != expected {
        t.Errorf("HelloWorld() = %v, want %v", observed, expected)
    }
}
```

Les tests - test

Un pattern très utilisé est le table-driven test.

👉 https://github.com/golang/go/wiki/TableDrivenTests

👉 https://dave.cheney.net/2019/05/07/prefer-table-driven-tests

Les tests - benchmark

Une méthode de benchmark doit commencer par **Benchmark**.

```
func BenchmarkHelloWorld(b *testing.B) {
    for i := 0; i < b.N; i++ {
        HelloWorld()
    }
}
```

go test -bench=HelloWorld ou **go test -bench=.**

Les tests - example

Une méthode example doit commencer par **Example.**

L'objectif de ces méthode est de servir de documentation vivante car le tooling de go est capable de générer un site web à partir du code en incluant des exemples executables.

```
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("laval"))
    fmt.Println(IsPalindrome("coucou"))
    // Output:
    // true
    // false
}
```

Les tests

Pour finir il est possible de faire du profiling à partir des tests.

go test -cpuprofile cpu.prof

go test -blockprofile bloc.prof

go test -memprofile mem.prof

go tool pprof cpu.prof

La concurrence

La concurrence est une préoccupation au coeur du langage. Il sait nativement gérer les coeurs d'un processeur.

Go repose sur un mécanisme de processus indépendant, les **goroutines**. Il est possible de gérer des milliers de goroutines sans problème.

Le processus **main** est lui-même une goroutine.

Les goroutines sont légères mais pas **gratuites** (2kb).

Conceptuellement, un scheduler est responsable de répartir les différentes goroutines sur les différents threads du host.

La concurrence

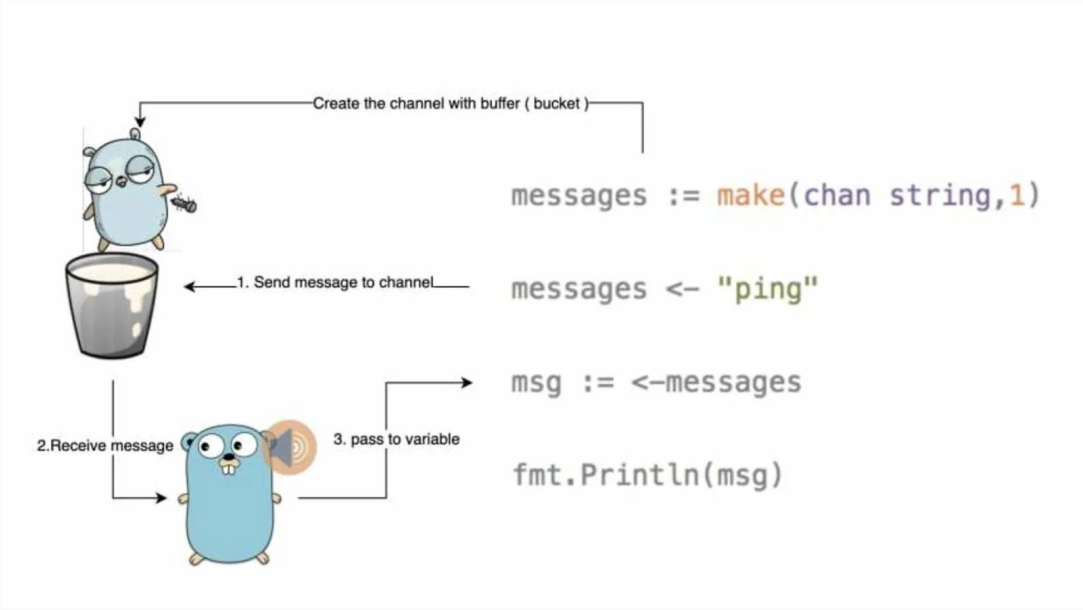
La concurrence en GO repose sur CSP (Communicating Sequential Processes) qui intègre un mécanisme de synchronisation basé sur le rendez-vous. CSP a été décrit par Hoare en 1978.

> “Don’t communicate by sharing memory, share memory by communicating.”
>
> Rob Pike

Il n’y a pas de mémoire partagée entre les goroutines mais un mécanisme de message permet aux goroutines de s’échanger des données, ce sont les **channels**.

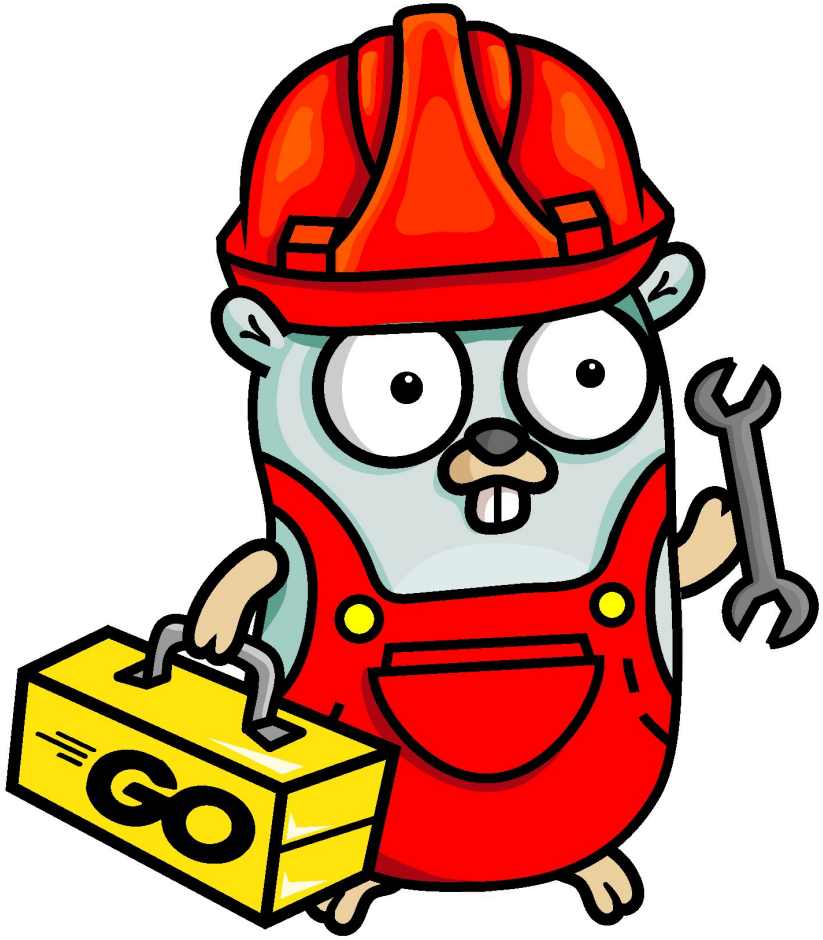
Pas besoin de mutex 🙂

La concurrence



https://dev.to/ahmedash95/understand-golang-channels-and-how-to-monitor-with-grafana-154

👉 [Au pays des Gophers] https://www.youtube.com/watch?v=DdW9ERd3iMk


GO

Advanced

- Layout des projets
- L’héritage
- L’escape analysis
- Alignement des variables
- Gestion des contrats
- Gestion des dépendances
- Simple design
- TDD
- Architecture hexagonale

Layout des projets

Privilégier un découpage simple et orienté fonctionnel. Comme par exemple :

- un package racine réservé pour les objets métiers
- un sous-package par dépendance à un tiers
- un package dédié aux objets pour les tests si nécessaire (mock)
- le main package pour relier toutes les dépendances

👉 https://tutorialedge.net/golang/go-project-structure-best-practices/

👉 https://github.com/hashicorp/consul/tree/main/api

👉 https://rakyll.org/style-packages/

Layout des projets

👉 https://dave.cheney.net/2019/10/06/use-internal-packages-to-reduce-your-public-api-surface

Si besoin de limiter la portée d'un package, il est possible de déclarer un package d'un répertoire **/internal**.

Seul les packages présents dans la même hiérarchie du package présent dans **/internal** pourront être importés.

L’héritage

Pas d’héritage en GO. Il faut donc utiliser la composition.

Il est possible d’inclure un struct dans un struct afin de profiter de ses attributes, c’est ce qu’on appelle le **struct embedding**.

```
type Point struct {
    X, Y int
}

type Circle struct {
    Point
    Radius int
}
```

L’héritage

Il est aussi possible d’inclure des **interface** ou des ***struct**.

👉 https://travix.io/type-embedding-in-go-ba40dd4264df

Dans une **interface**:

- on ne peut inclure que des **interfaces** 🙂
- les méthodes doivent être différentes

Dans un **struct**:

- on peut inclure des **interfaces** et des **structs**
- en cas de conflit, c’est le champ ou la méthode de “moindre profondeur” qui prévaut (“promotion”)

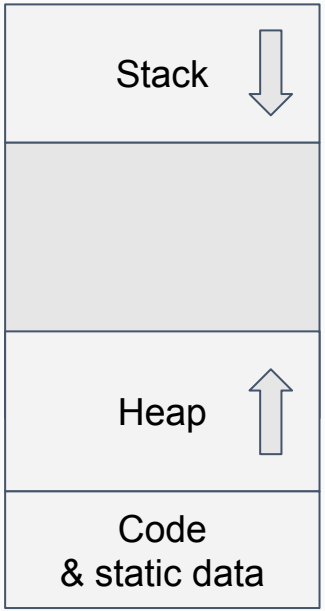
L'escape analysis

Pour chacune des variables d'un programme, le compilateur Go va choisir d'allouer cette variable sur la Heap ou sur la Stack. Cette action est appelée **escape analysis**.

L'allocation sur la stack est négligeable tandis que l'allocation sur la heap est chère.

Pour utiliser la Stack seules 2 instructions CPU sont nécessaires. La lecture et l'écriture sur la Heap est bien plus complexe.

Autre point à noter, le GC ne passe que sur la Heap.



L’escape analysis

Le compilateur construit un arbre d’appels de fonctions et trace le flux des arguments en entrée et des valeurs en retour. Pour prendre connaissance des choix du compilateur :

go build -gcflags '-m'

Voici des patterns connus qui entraînent une allocation vers la Heap :

- envoyer des pointers ou des valeurs contenant des pointers à un chanel
- stocker des pointers ou des valeurs contenant des pointers dans un slice
- augmenter la capacité d’un slice au runtime
- appeler une méthode via son interface

A garder à l’esprit :

- les pointers pointent sur des données allouées sur la Heap
- map et channel sont des pointers de Type runtime
- slice et interface sont des containers qui possèdent respectivement 1 et 2 pointers
- **Pas d’optimisation prématurée !**

L’alignement des variables

Afin d’optimiser la mémoire, il est possible d’ordonner la déclarations des variables afin de limiter ou de supprimer le padding mémoire lié à l’alignement des variables.

“La règle généralement applicable pour qu'une donnée soit bien alignée, est qu'elle se trouve à **une adresse divisible par sa taille**. Ainsi, une donnée occupant un seul octet est toujours bien alignée, une donnée de deux octets est bien alignée si elle est à une adresse paire, une donnée de 4 octets est bien alignée si elle est à une adresse divisible par 4, etc.

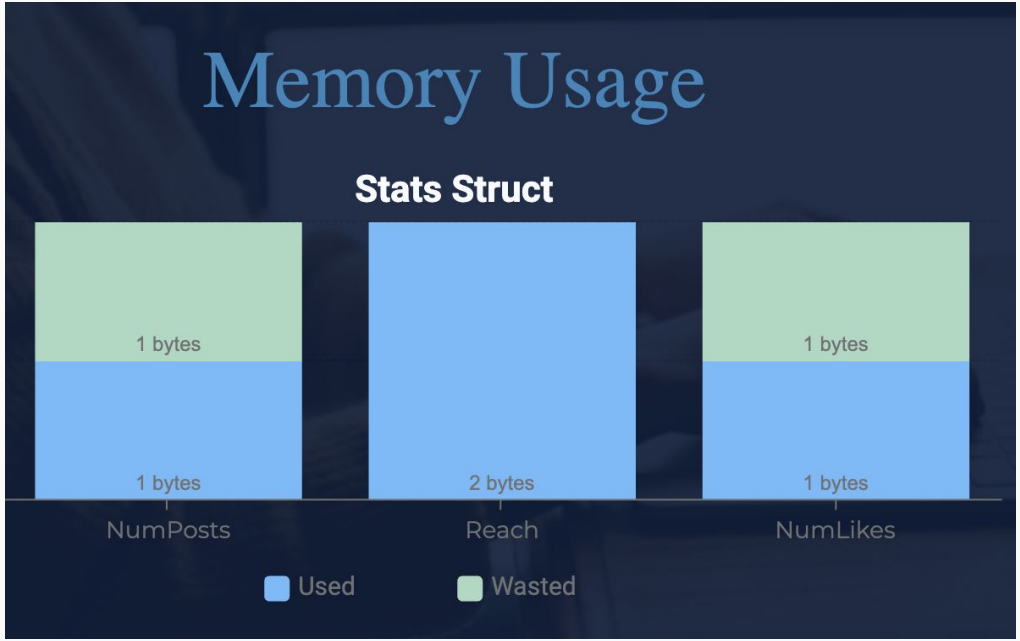
Les contraintes d'alignement en mémoire dépendent de l'architecture du processeur. “

https://wagslane.dev/posts/go-struct-ordering/

L’alignement des variables

Voici un exemple :

```
type stats struct {
    NumPosts uint8
    Reach    uint16
    NumLikes uint8
}
```

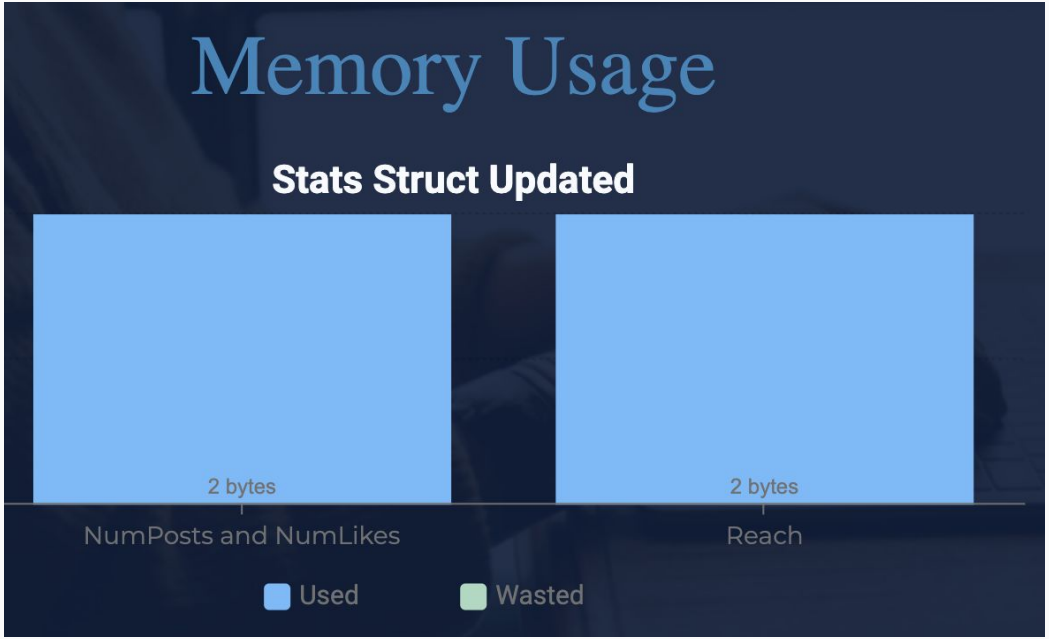


https://wagslane.dev/posts/go-struct-ordering/

L’alignement des variables

Si on change l’ordre de déclaration des variables, nous pouvons réduire la consommation mémoire.

```
type stats struct {
    Reach    uint16
    NumPosts uint8
    NumLikes uint8
}
```



https://wagslane.dev/posts/go-struct-ordering/

Il est important de noter que l’impact sur la mémoire est dépendant du nombre d’objets qui seront créés.

Gestion des contrats

Les contrats sont gérés par des **interfaces**.

Elles servent à définir un comportement souhaité, il est donc préférable qu'elle soit définie du côté de l'appelant.

Il y a plusieurs avantages à cette pratique :

- diminue le couplage entre l'appelant et l'appelé (couplage faible 😍)
- le contrat est réduit à son propre besoin (Interface Segregation Principle)

Le structural typing nous assure que le contrat sera respecté dès la compilation.

Gestion des dépendances

Avant la version 1.11 de GO, la gestion des dépendances était “simpliste”. Les dépendances étaient gérées via le clone de projets sans gestion de version.

Le projet **vgo**, portait par **Russ Cox**, était une des solutions pour gérer ce problème (support du semver).

Gestion des dépendances

Le projet vgo a tellement bien marché qu'il a été intégré dans le tooling de GO 1.11 (en experimental) via la commande **go mod**. Les modules deviennent "ready for production" à partir de GO 1.14.

Un module est un ensemble de packages qui sont versionnés ensemble. Ils sont vu comme une seule unité. Un repository pourra définir de 1 à n modules.

L'utilisation de modules permet d'avoir des builds fiables et reproductibles. Les versions des dépendances sont explicites.

Gestion des dépendances

```
go mod init github.com/my/repo
```

Va permettre de créer un fichier **go.mod** et un fichier **go.sum** (ensemble de hash pour chaque dépendance, ce n'est pas un **lock file**)

```
// exemple de fichier go.mod
module github.com/my/thing

go 1.17

require (
    github.com/some/dependency v1.2.3
    github.com/another/dependency/v4 v4.0.0
)
```

Gestion des dépendances

go build, go test permettent d'ajouter les dépendances nécessaires dans le fichier go.mod.

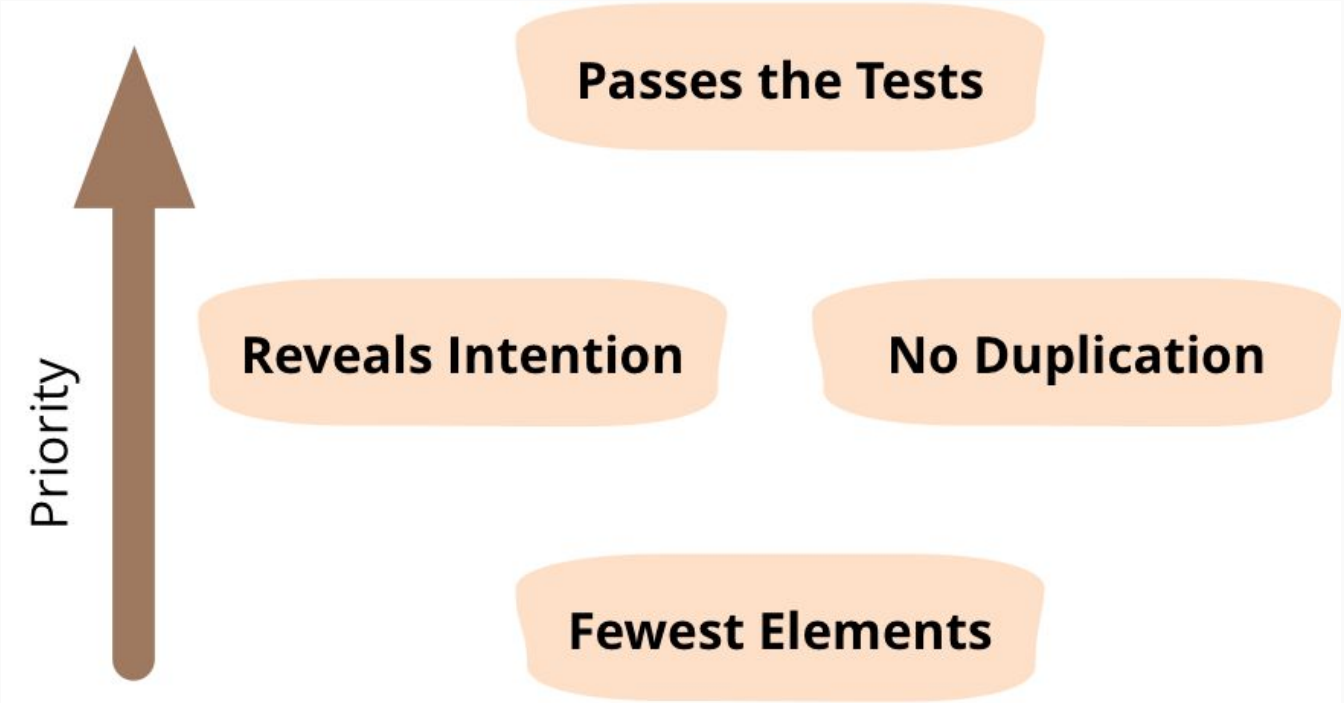
go list -m all est une commande affichant l'ensemble des dépendances.

go get [ma dépendance] va changer la version d'une dépendance ou en ajouter une nouvelle.

go mod tidy va supprimer les dépendances qui ne sont plus utilisées.

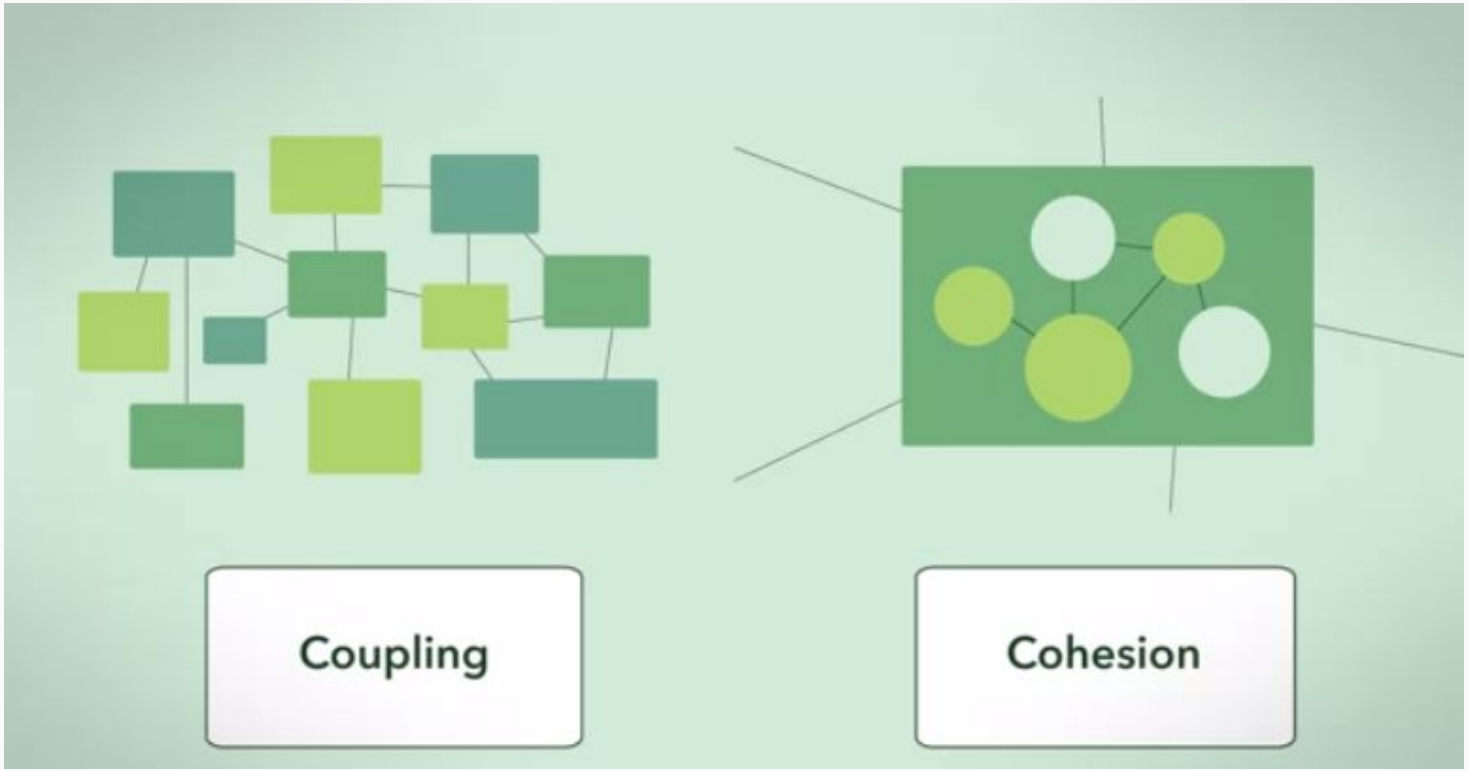
👉 https://blog.golang.org/using-go-modules
👉 https://github.com/golang/go/wiki/Modules
👉 https://golang.org/ref/mod

Simple design



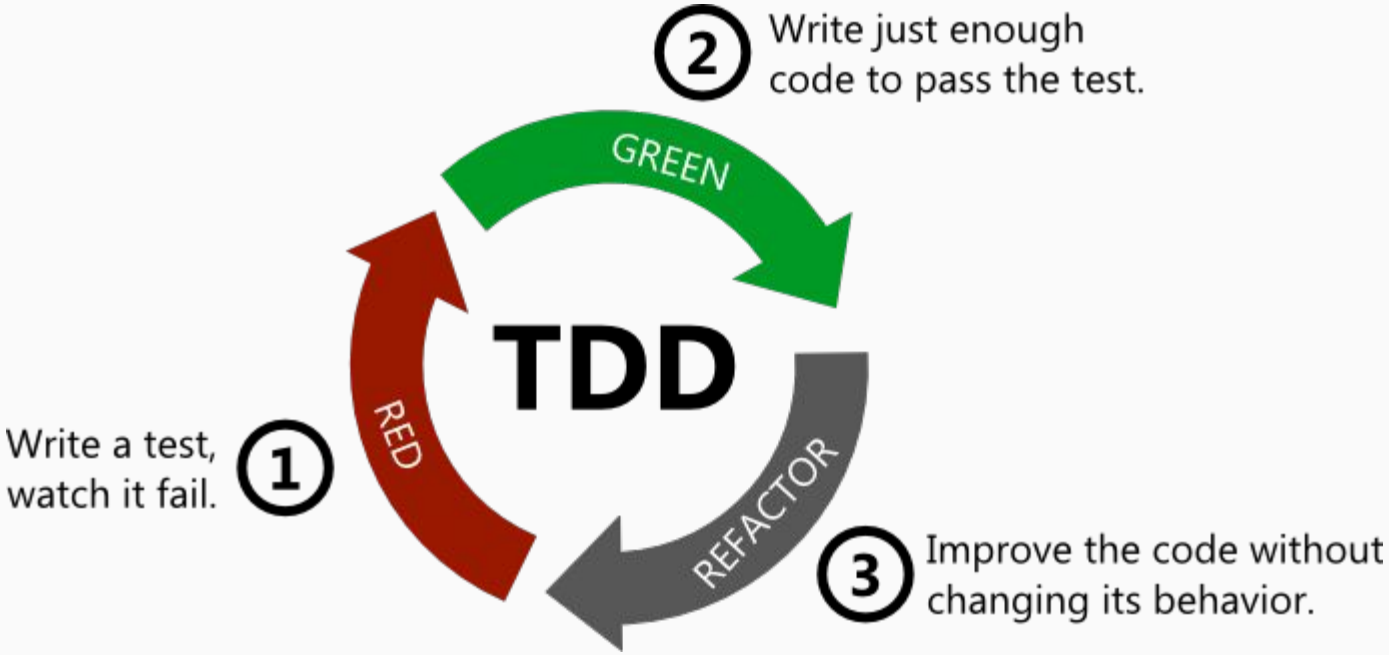
Les 4 règles du Simple design de Kent Beck

Cohésion et couplage



https://www.coursera.org/lecture/object-oriented-design/1-3-1-coupling-and-cohesion-q8wGt

Test Driven Development

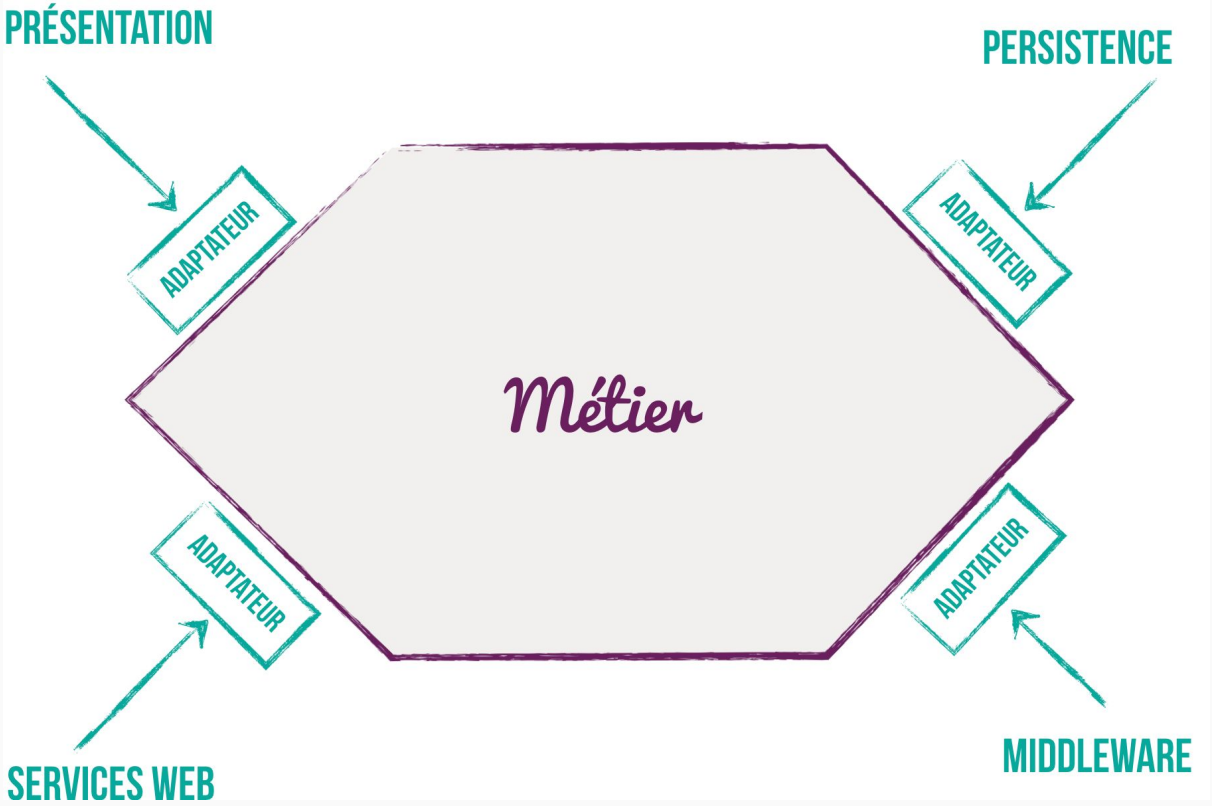


Test Driven Development

The Three Rules of TDD d'Uncle Bob

- You may not write production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- You may not write more production code than is sufficient to pass the currently failing test.

Architecture hexagonale



https://blog.engineering.publicissapient.fr/2016/03/16/perennisez-votre-metier-avec-larchitecture-hexagonale/

Pratiquer

Advent of code 👉 https://adventofcode.com/

Exercism 👉 https://exercism.io/my/tracks

Coding Game 👉 https://www.codingame.com/training

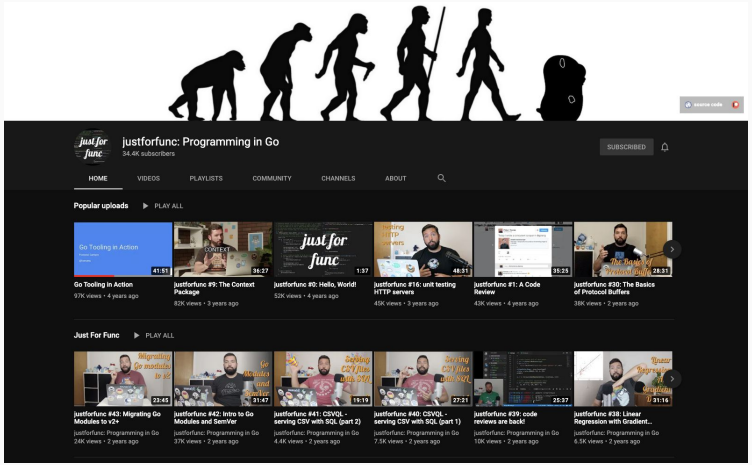
Hash Code 👉 https://codingcompetitions.withgoogle.com/hashcode/

Références

- https://blog.golang.org/
- https://dave.cheney.net/practical-go
- https://golang.org/doc/effective_go.html

- https://play.golang.org/

- https://github.com/stretchr/testify



Just4Func

👉 https://www.youtube.com/channel/UC_BzFbxG2za3bp5NRRRXJSw

Projet - Gestion d’un entrepôt