Introduction au CI/CD

ENSG - Mars 2024

- Présentation disponible à l'adresse: https://cicd-lectures.github.io/slides/2024
- Version PDF de la présentation : Cliquez ici
- This work is licensed under a Creative Commons Attribution 4.0 International License
- Code source de la présentation: https://github.com/cicd-lectures/slides

Comment utiliser cette présentation ?

- Pour naviguer, utilisez les flèches en bas à droite (ou celles de votre clavier)
  - Gauche/Droite: changer de chapitre
  - Haut/Bas: naviguer dans un chapitre
- Pour avoir une vue globale : utilisez la touche "o" (pour "**Overview**")
- Pour voir les notes de l’auteur : utilisez la touche "s" (pour "**Speaker** notes")

Bonjour !

Damien DUPORTAL

- Staff Software Engineer chez CloudBees pour le projet Jenkins 👨🏻‍⚖️
- Freelancer
- Me contacter :
  - damien.duportal <chez> gmail.com
  - dduportal
  - Damien Duportal
  - @DamienDuportal

Julien LEVESY

- Senior Platform Engineer @ Voi 🛴
- Me contacter :
  - jlevesy <chez> gmail.com
  - Julien Levesy
  - @jlevesy

Et vous ?

A propos du cours

- Alternance de théorie et de pratique pour être le plus interactif possible
- Reproductible à la maison, pensé dans le contexte du "Covid à la maison"
- Contenu entièrement libre et open-source. Le code source est visible sur cette page
- L’évaluation sera faite sur votre projet final. Des indications vous seront données en fin de cours!

Avant de commencer

- Comment souhaitez vous gérer les pauses?

Comment mener un projet logiciel?

Avant : le cycle en V

Que peut-il mal se passer?

- On spécifie et l'on engage un volume conséquent de travail sur des hypothèses
  - … et si les hypothèses sont fausses?
  - … et si les besoins changent?
- Cycle trèèèèès long
  - Aucune validation à court terme
  - Coût de l'erreur décuplé

Comment éviter ça?

- Valider les hypothèses au plus tôt, et étendre petit à petit le périmètre fonctionnel.
  - Réduire le périmètre fonctionnel au minimum.
  - Confronter le logiciel au plus tôt aux utilisateurs.
  - Refaire des hypothèses basées sur ce que l'on à appris, et recommencer!
- "Embrasser" le changement
  - Votre logiciel va changer en **continu**

La clé : gérer le changement!

- Le changement ne doit pas être un événement, ça doit être la norme.
- Notre objectif : minimiser le coût du changement.
- Faire en sorte que:
  - Changer quelque chose soit facile
  - Changer quelque chose soit rapide
  - Changer quelque chose ne casse pas tout

Heureusement, vous avez des outils à disposition!

Et c’est ce que l’on va voir ensemble pendant les trois prochains jours!

Préparer votre environnement de travail

Outils Nécessaires 🛠️

- Un navigateur récent (et décent)
- Un compte sur GitHub
- On va vous demander de travailler en binôme, commencez à réfléchir avec qui vous souhaitez travailler !
- Enregistrez vous par ici!

GitPod

GitPod.io : Environnement de développement dans le ☁ "nuage"

- **But:** reproductible
- Puissance de calcul sur un serveur distant
- Éditeur de code VSCode dans le navigateur
- Gratuit pour 50h par mois (⚠)
- Open-Source : vous pouvez l'héberger chez vous

Démarrer avec GitPod 🚀

- Rendez vous sur https://gitpod.io
- Authentifiez vous en utilisant votre compte GitHub:
  - Bouton "Login" en haut à droite
  - Puis choisissez le lien "Continue with GitHub"

⚠ Pour les "autorisations", passez sur la slide suivante

Autorisations demandées par GitPod 🔓

Lors de votre première connexion, GitPod va vous demander l'accès (à accepter) à votre email public configuré dans GitHub :

⚠ Passez à la slide suivante avant d'aller plus loin

Validation du Compte GitPod 📱

GitPod vous demande votre numéro de téléphone mobile afin d'éviter les abus (service gratuit).
Saisissez un numéro de téléphone valide pour recevoir par SMS un code de déblocage :

⚠ Passez à la slide suivante avant d'aller plus loin

Choisissez l'éditeur "VSCode Browser" (la première tuile) :

Workspaces GitPod 🗂️

- Vous arrivez sur la page listant les "workspaces" GitPod :
- Un workspace est une instance d'un environnement de travail virtuel (C'est un ordinateur distant)
- ⚠ Faites attention à réutiliser le même workspace tout au long de ce cours⚠

Permissions GitPod ↔ GitHub 🔐

- Pour les besoins de ce cours, vous devez autoriser GitPod à pouvoir effectuer certaines modification dans vos dépôts GitHub
- Rendez-vous sur la page des intégrations avec GitPod
- Éditez les permissions de la ligne "GitHub" (les 3 petits points à droits) et sélectionnez uniquement :
  - `user:email`
  - `public_repo`
  - `workflow`

Démarrer l'environnement GitPod

Cliquez sur le bouton ci-dessous pour démarrer un environnement GitPod personnalisé:

Open in Gitpod

Après quelques secondes (minutes?), vous avez accès à l'environnement:

- Gauche: navigateur de fichiers ("Workspace")
- Haut: éditeur de texte ("Get Started")
- Bas: Terminal interactif
- À droite en bas: plein de popups à ignorer (ou pas?)

Source disponible dans: https://github.com/cicd-lectures/gitpod

Checkpoint 🎯

- Vous devriez pouvoir taper la commande `whoami` dans le terminal de GitPod:
  - Retour attendu: `gitpod`
- Vous devriez pouvoir fermer le fichier "Get Started"…
  - … et ouvrir le fichier `.gitpod.yml`

On peut commencer !

Guide de survie en ligne de commande

Remise à niveau / Rappels

CLI

- 🇬🇧 CLI == "Command Line Interface"
- 🇫🇷 "Interface de Ligne de Commande"

Anatomie d’une commande

```
ls --color=always -l /bin
```

Copy

- Séparateur : l’espace
- Premier élément (`ls`) : c’est la commande
- Les éléments commençant par un tiret – sont des "options" et/ou drapeaux ("flags")
  - "Option" == "Optionnel"
- Les autres éléments sont des arguments (`/bin`)
  - Nécessaire (par opposition)

Manuel des commande

- Afficher le manuel de `<commande>` :

```
man <commande> # Commande 'man' avec comme argument le nom de ladite commande
```

  - Navigation avec les flèches haut et bas
    - Tapez `/` puis une chaîne de texte pour chercher
    - Touche `n` pour sauter d'occurrence en occurrence
  - Touche `q` pour quitter

🎓 Essayez avec `ls`, cherchez le mot `color`

- 💡 La majorité des commandes fournit également une option (`--help`), un flag (`-h`) ou un argument (`help`)

Raccourcis

Dans un terminal Unix/Linux/WSL :

- `CTRL + C` : Annuler le processus ou prompt en cours
- `CTRL + L` : Nettoyer le terminal
- `CTRL + A` : Positionner le curseur au début de la ligne
- `CTRL + E` : Positionner le curseur à la fin de la ligne
- `CTRL + R` : Rechercher dans l'historique de commandes
- `Tab`: Compléter la commande

🎓 Essayez-les !

- `pwd` : Afficher le répertoire courant
  - 🎓 Option `-P` ?
- `ls` : Lister le contenu du répertoire courant
  - 🎓 Options `-a` et `-l` ?
- `cd` : Changer de répertoire
  - 🎓 Sans argument : que se passe t'il ?
- `cat` : Afficher le contenu d'un fichier
  - 🎓 Essayez avec plusieurs arguments
- `mkdir` : créer un répertoire

Commandes de base 2/2

- `echo` : Afficher un (des) message(s)
- `rm` : Supprimer un fichier ou dossier
- `touch` : Créer un fichier
- `grep` : Chercher un motif de texte
- `code` : Ouvre le fichier dans l'éditeur de texte

- Le système de fichier a une structure d'arbre
  - La racine du disque dur c'est `/` : 🎓 `ls -l /`
  - Le séparateur c'est également `/` : 🎓 `ls -l /usr/bin`
- Deux types de chemins :
  - Absolu (depuis la racine): Commence par `/` (Ex. `/usr/bin`)
  - Sinon c'est relatif (e.g. depuis le dossier courant) (Ex `./bin` ou `local/bin/`)

Arborescence de fichiers 2/2

- Le dossier "courant" c'est `.` : 🎓 `ls -l ./bin # Dans le dossier /usr`
- Le dossier "parent" c'est `..` : 🎓 `ls -l ../ # Dans le dossier /usr`
- ~ (tilde) c'est un raccourci vers le dossier de l'utilisateur courant : 🎓 `ls -l ~`
- – (minus) raccourci pour revenir au dernier répertoire visité
- Sensible à la casse (majuscules/minuscules) et aux espaces : 🎓

```
ls -l /bin
ls -l /Bin
mkdir ~/"Accent tué"
ls -d ~/Accent\ tué
```

Copy

Un language (?)

- Variables interpolées avec le caractère "dollar" `$` :

```
echo $MA_VARIABLE
echo "$MA_VARIABLE"
echo ${MA_VARIABLE}

# Recommendation
echo "${MA_VARIABLE}"

MA_VARIABLE="Salut tout le monde"

echo "${MA_VARIABLE}"
```

- Sous commandes avec `$(<command>)` :

```
echo ">> Contenu de /tmp :\n$(ls /tmp)"
```

- Des `if`, des `for` et plein d'autres trucs (https://tldp.org/LDP/abs/html/)

Codes de sortie

- Chaque exécution de commande renvoie un code de retour (🇬🇧 "exit code")
  - Nombre entier entre 0 et 255 (en POSIX)
- Code accessible dans la variable **éphémère** `$?` :

```
ls /tmp
echo $?

ls /do_not_exist
echo $?

# Une seconde fois. Que se passe-t'il ?
echo $?
```

Copy

Entrée, sortie standard et d'erreur

```
ls -l /tmp
echo "Hello" > /tmp/hello.txt
ls -l /tmp
ls -l /tmp >/dev/null
ls -l /tmp 1>/dev/null

ls -l /do_not_exist
ls -l /do_not_exist 1>/dev/null
ls -l /do_not_exist 2>/dev/null

ls -l /tmp /do_not_exist
ls -l /tmp /do_not_exist 1>/dev/null 2>&1
```

Copy

Pipelines

- Le caractère "pipe" `|` permet de chaîner des commandes
  - Le "stdout" de la première commande est branchée sur le "stdin" de la seconde
- Exemple : Afficher les fichiers/dossiers contenant le lettre `d` dans le dossier `/bin` :

```
ls -l /bin

ls -l /bin | grep "d" --color=auto
```

Copy

Exécution 1/2

- Les commandes sont des fichier binaires exécutables sur le système :

```
command -v cat # équivalent de "which cat"

ls -l "$(command -v cat)"
```

Copy

- La variable d'environnement `$PATH` liste les dossiers dans lesquels chercher les binaires
  - 💡 Utiliser cette variable quand une commande fraîchement installée n'est pas trouvée

Exécution 2/2

- Exécution de scripts :
  - Soit appel direct avec l'interpréteur : `sh ~/monscript.txt`
  - Soit droit d'exécution avec un "shebang" (e.g. `#!/bin/bash`)

```
$ chmod +x ./monscript.sh

$ head -n1 ./monscript.sh
#!/bin/bash

$ ./monscript.sh
# Exécution
```

Copy

Comment fonctionnent les Internets?

🧐 Que se passe-t-il quand je tape google.com dans mon navigateur et que j'appuie sur entrée?

1. 📖 Resolution DNS
2. 🛜 Connection TCP
3. 🔒 Handshake TLS
4. ➡ Envoi d’une requête HTTP au Serveur
5. ⬅ Réception d’une reponse et décodage du contenu
6. 🎨 Rendu de la page

Zoom sur HTTP

- Hypertext Transfer Protocol
- Définit un format de requête/réponse dans le modèle client / serveur
  - Le client demande une ressource à un serveur via une requête HTTP, serveur lui réponds une réponse avec un contenu.
- Plusieurs versions en activité HTTP/1.1, HTTP/2, HTTP/3, mais la sémantique reste la même

Anatomie d'une requête HTTP

Une requête est composée des champs suivant:

- **Méthode**: Indique une action désirée (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`...)
- **Hote**: indique un domaine dans lequel récupérer les resources (`github.com`)
- **Chemin** (path): indique une ressource à obtenir au serveur (`/assets/file.js`)
- **Paramètres de requête** (query parameters): paramètres additionnels de requête apposés au path (`/pages/node?utm_source=facebook`)
- **Entêtes** (headers): Couple clé → multiples valeurs indiquant des méta information sur la requête (`Accepted-Content`, `User-Agent`,`Accept`, `Referrer`, `Authorization`, `Cookies`)
- **Corps** (body): Optionnel, contenu encodé à envoyer au serveur, par exemple une soumission de formulaires

Anatomie d'une réponse HTTP

Une réponse est composée des champs suivant:

- D'un status code 🐱
  - 200 OK, 404 Not Found, 301 Moved Permanently etc..
- **Entêtes** (headers): Couple clé → multiples valeurs indiquant des méta information sur la réponse (`Content-Length`, `Content-Encoding`,`Content-Type` ...)
- **Un corps de réponse** à lire et à décoder

Comment parler HTTP depuis le terminal?

- On propose d'utiliser cURL
- Outil pour transférer des données dans différents protocoles
  - Le couteau suisse des internets!

🎓 Exercice: Première Requête en utilisant cURL

- Que signifie cette ligne de commande?
  - Indice: `man curl`
- Que pouvez vous dire du résultat affiché?

```
curl --verbose --location --output /dev/null voi.com
```

Copy

✅ Solution: Première Requête en utilisant cURL (1/4)

- C'est verbeux 🙃, mais on l'a demandé avec `--verbose`. cURL va logger tous les échanges effectués avec le serveur
- `--location` indique à cURL de suivre les redirections
- `--output` indique à cURL d'écrire le contenu dans répondu `/dev/null` au lieu de l'afficher sur la sortie standard

✅ Solution: Première Requête en utilisant cURL (2/4)

Regardons d'un peut plus près les logs:

```
# On se connecte a une IPv6... probablement celle de voi.com?
* Trying [2606:4700:20::681a:3d6]:80...
* Connected to voi.com (2606:4700:20::681a:3d6) port 80

# cURL formule la requête demandée sur HTTP.
> GET / HTTP/1.1
> Host: voi.com
> User-Agent: curl/8.4.0
> Accept: */*
>
# Le serveur nous réponds une 301 !? voi.com à bougé?
< HTTP/1.1 301 Moved Permanently
# [...]
# Aha! Le serveur nous redirige vers le même site, mais en HTTPS sur le port 443.
< Location: https://voi.com:443/
```

Copy

cURL (3/4)

```
# Comme indiqué: on se reconnecte a voi.com sur le port 443!
* Clear auth, redirects to port from 80 to 443
* Issue another request to this URL: 'https://voi.com:443/'
*   Trying [2606:4700:20::681a:3d6]:443...
* Connected to voi.com (2606:4700:20::681a:3d6) port 443

# On se connecte en HTTPS, du coup il va falloir établir une session TLS
# Ensuite cURL et le serveur se mettent d'accord et établissent la connexion sécurisée.
* (304) (OUT), TLS handshake, Client hello (1):
# [...]
# On est connectés de façon sécurisée au serveur!
* SSL connection using TLSv1.3 / AEAD-CHACHA20-POLY1305-SHA256
* Server certificate:
# [...] Le certificat du serveur est valide!
*  SSL certificate verify ok.
# [...] On refait notre requête une fois connectés!
> GET / HTTP/2
> Host: voi.com
> User-Agent: curl/8.4.0
> Accept: */*
>
# Victoire le serveur nous réponds!
< HTTP/2 200
```

Copy

✅ Solution: Première Requête en utilisant cURL (4/4)

- Ce qu'il viens de se passer est ce que l'on appelle une `HTTPS` upgrade
- Le serveur force le client a se connecter de façon sécurisée!
- Pourquoi?
  - TLS prouve que le client parle bien au bon serveur!
  - TLS chiffre les communications sur le réseau, on peut faire transiter des données sans(trop) se soucier d'être espionnés 🕵️

- Maintenant essayez d'enlever l'option `--location`, que se passe-t-il?
- Maintenant essayez d'enlever l'option `--output /dev/null`, que se passe-t-il?

Autres Options Utiles de cURL

- Contrôle de la méthode de la requête: `--request POST`, `--request DELETE`
- Ajouter un header a la requête: `--header "Content-Type: application/json"`
- Envoyer un body dans la requête:
  - Directement depuis la ligne de commande `--data '{"some":"json"}`
  - En lisant un ficher `--data '@some/local/file'`

Essayez donc sur voi.com!

🎓 Exercice: Afficher du JSON de Façon Lisible

- Qu'affiche le résultat de la commande suivante?
- Comment le rendre plus lisible?
  - Indice: il faut utilser un `|` (pipe) et la commande `jq`

```
curl https://swapi.dev/api/planets/1
```

Copy

✅ Solution: Afficher du JSON de Façon Lisible

```
curl https://swapi.dev/api/planets/1 | jq .
```

Copy

- Bonus: jq permets de sélectionner un attribut JSON.

```
curl https://swapi.dev/api/planets/1 | jq .residents
```

Copy

Checkpoint 🎯

- Internet repose sur une collection de protocole (DNS, TCP, TLS, HTTP)
- HTTP permets de formuler une requête à un serveur et une réponse
- `cURL` est un outil très complet pour parler HTTP depuis un terminal!

Les fondamentaux de git

Tracer le changement dans le code

avec un **VCS** : 🇬🇧 Version Control System

également connu sous le nom de SCM (🇬🇧 Source Code Management)

Pourquoi un VCS ?

- Pour conserver une trace de **tous** les changements dans un historique
- Pour **collaborer** efficacement sur un même référentiel de code source

Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1

Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Computer B
File
Version Database

Quel VCS utiliser ?

Nous allons utiliser **Git**

Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

https://git-scm.com/

Les 3 états avec Git

- L'historique ("Version Database") : dossier `.git`
- Dossier de votre projet ("Working Directory") - Commande
- La zone d'index ("Staging Area")

Source : https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Rudiments-de-Git#*les_trois*%C3%A9tats

🎓 Exercice :avec Git - 1.1

- Dans le terminal de votre environnement GitPod:
  - Créez un dossier vide nommé `projet-vcs-1` dans le répertoire `/workspace`, puis positionnez-vous dans ce dossier

    ```
    mkdir -p /workspace/projet-vcs-1/
    cd /workspace/projet-vcs-1/
    ```

    Copy

  - Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?
- Initialisez le dépôt git avec `git init`
  - Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?

✅ Solution : avec Git - 1.1

```
mkdir -p /workspace/projet-vcs-1/
cd /workspace/projet-vcs-1/
ls -la # Pas de dossier .git
git status # Erreur "fatal: not a git repository"
git init ./
ls -la # On a un dossier .git
git status # Succès avec un message "On branch master No commits yet"
```

Copy

🎓 Exercice :avec Git - 1.2

- Créez un fichier `README.md` dedans avec un titre et vos nom et prénoms
  - Essayez la commande `git status` ?
- Ajoutez le fichier à la zone d'indexation à l'aide de la commande `git add (...)`
  - Essayez la commande `git status` ?
- Créez un commit qui ajoute le fichier `README.md` avec un message, à l'aide de la commande `git commit -m <message>`
  - Essayez la commande `git status` ?

✅ Solution : avec Git - 1.2

```
echo "# Read Me\n\nObi Wan" > ./README.md
git status # Message "Untracked file"

git add ./README.md
git status # Message "Changes to be committed"
git commit -m "Ajout du README au projet"
git status  # Message "nothing to commit, working tree clean"
```

Copy

diff: un ensemble de lignes "changées" sur un fichier donné

10 cluster/addons/node-problem-detector/npd.yaml

```diff
@@ -26,28 +26,28 @@ subjects:
 26  apiVersion: apps/v1
 27  kind: DaemonSet
 28  metadata:
-29    name: npd-v0.8.0
+29    name: npd-v0.8.5
 30    namespace: kube-system
 31    labels:
 32      k8s-app: node-problem-detector
-33      version: v0.8.0
+33      version: v0.8.5
 34      kubernetes.io/cluster-service: "true"
 35      addonmanager.kubernetes.io/mode: Reconcile
 36  spec:
 37    selector:
 38      matchLabels:
 39        k8s-app: node-problem-detector
-40        version: v0.8.0
+40        version: v0.8.5
 41    template:
 42      metadata:
 43        labels:
 44          k8s-app: node-problem-detector
-45          version: v0.8.0
+45          version: v0.8.5
 46          kubernetes.io/cluster-service: "true"
```

changeset: un ensemble de "diff" (donc peut couvrir plusieurs fichiers)

Showing **12 changed files** with **314 additions** and **200 deletions**.

3 Jenkinsfile

10 make.ps1

456 tests/plugins-cli.Tests.ps1

commit: un changeset qui possède un (commit) parent, associé à un message

"HEAD": C'est le dernier commit dans l'historique

🎓 Exercice :avec Git - 2

- Afficher la liste des commits
- Afficher le changeset associé à un commit
- Modifier du contenu dans `README.md` et afficher le diff
- Annulez ce changement sur `README.md`

✅ Solution : avec Git - 2

```
git log

git show # Show the "HEAD" commit
echo "# Read Me\n\nObi Wan Kenobi" > ./README.md

git diff
git status

git restore README.md
git status
```

Copy

Terminologie de Git - Branche

- Abstraction d’une version "isolée" du code
- Concrètement, une **branche** est un alias pointant vers un "commit"

🎓 Exercice :avec Git - 3

- Créer une branche nommée `feature/html`
- Ajouter un nouveau commit contenant un nouveau fichier `index.html` sur cette branche
- Afficher le graphe correspondant à cette branche avec `git log --graph`

✅ Solution : avec Git - 3

```
git branch feature/html && git switch feature/html
# Ou git switch --create feature/html
echo '<h1>Hello</h1>' > ./index.html
git add ./index.html && git commit --message="Ajout d'une page HTML par défaut" # -m / --message

git log
git log --graph
git lg # cat ~/.gitconfig => regardez la section section [alias], cette commande est déjà définie!
```

Copy

- On intègre une branche dans une autre en effectuant un **merge**
- Plusieurs strategies sont possibles pour merger:
  - Quand l'historique de commit n'a pas diverge: git fait avancer la branche directement, c'est un **fast-forward**
  - Dans le cas contraire, un nouveau commit est créé, fruit de la combinaison de 2 autres commits

🎓 Exercice :avec Git - 4

- Merger la branche `feature/html` dans la branche principale
  - ⚠ Pensez à utiliser l'option `--no-ff` (no fast forward) pour forcer git a créer un commit de merge.
- Afficher le graphe correspondant à cette branche avec `git log --graph`

✅ Solution : avec Git - 4

```
git switch main
git merge --no-ff feature/html # Enregistrer puis fermer le fichier 'MERGE_MSG' qui a été ouvert
git log --graph

# git lg
```

Copy

Exemple d’usages de VCS

- "Infrastructure as Code" :
  - Besoins de traçabilité, de définition explicite et de gestion de conflits
  - Collaboration requise pour chaque changement (revue, responsabilités)
- Code Civil:
  - https://github.com/steeve/france.code-civil
  - https://github.com/steeve/france.code-civil/pull/40
  - https://github.com/steeve/france.code-civil/commit/b805ecf05a86162d149d3d182e04074ecf72c066

🎯 Checkpoint

On a vu :

- A quoi sert `git` et sa nomenclature de base (diff, changest, commit, branch)
- A quoi reconnaître un dépôt initialisé local et l'espace de travail associé
- Comment utiliser git localement (ajouter au staging, commiter)
- l'historique et un merge avec git (localement)

065 五 一

Pourquoi Go dans ce Cours?

- Peu importe les outils, les problèmes sous jacents lié au CI/CD sont les mêmes
- Faire face à un nouveau langage est quelque chose de fréquent
- Beaucoup d’opportunités professionnelles autour de ce langage

Qu’est ce que Go?

- 🧱 Langage fortement typé
- 💻 Compilé
- © Syntaxe proche du C
- 🌥 Gestion de la mémoire automatisée
- 🔄 Conçu pour le traitement concurrent

Go Propulse Le Cloud!

- Issu de chez Google
- Première version publique en 2009
- v1.0 en 2012 … et rétrocompatible depuis!
- Utilisé dans de nombreux projets!
  - Docker, Kubernetes, Terraform, Prometheus, Grafana…

🎓 Exercice: Un Premier Programme en Go

- Dans le répertoire `workspace` créez un repertoire `helloworld`
- Dans ce répertoire, créez un fichier `main.go` et copiez le code ci-dessous.

```go
package main

import (
	"fmt"
)

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

- Compilez votre programme a l'aide de la commande `go build ./main.go`
- Executez le programme généré `./main`

✅ Solution: Un Premier Programme en Go

Copy

```
# Crée un répertoire helloworld
mkdir -p /workspace/helloworld
# Saute dans le répertoire
cd /workspace/helloworld
# Crée un fichier main.go
touch main.go
# Ouvre le fichier main.go dans l'éditeur
code main.go
# Compile le programme
go build ./main.go
# Exécute le programme.
./main
```

💡 `go run ./main.go` compile et exécute le programme directement!

⚠ Ne génère pas de binaire utilisable.

```
$ go run ./main.go
Hello ENSG!
```

Copy

Formatage de Code

- Formatage du code automatique, si vous appuyez sur `Ctrl+S`
- Pas de débat tabs vs spaces 🤡, c'est des tabs.

Anatomie d'un Fichier go (1/2)

```go
package main

import (
	"fmt"
)

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

Copy

Anatomie d’un Fichier go (2/2)

- `func`: Déclare une fonction `main`:
  - Cette fonction appelle la fonction `Println` du package `fmt`
  - En passant la chaine de caractères "Hello ENSG!"
- `package`: Déclare que ce fichier fait partie du package `main`.
- `import`: importe le package `fmt` dans le fichier

Packages & Imports (1/2)

- Un package est un groupe logique de symboles (variables, constantes, types et fonction)
- Un package est identifié par une URL indiquant ou le télécharger
  - 👉 Tous les packages sont uniques!
  - Ex: `github.com/jlevesy/prometheus-elector/config`
- Sauf pour la librairie standard, ou il n'y a pas de domaines
  - Ex: `net/http`, `os`
- Un package est importé par un autre package
  - Ici notre programme importe le package `fmt`

Packages & Imports (2/2)

- Un package est représenté par un répertoire dans un dépôt de code
- Tous les fichiers go présents dans un même répertoire doivent déclarer le même package
- **Convention**: Le nom du package déclaré est le même que celui du répertoire content le fichier
  - Ex: Les fichiers présents dans `github.com/jlevesy/prometheus-elector/config` commencent tous par `package config`

Mais ce n’est pas le cas de notre programme?

ℹ Le Package Spécial `main`

- Ce package est le `point d'entrée` du programme.
- La fonction `main` du package `main` est la première fonction appelée lors de l'exécution d'un programme go.

- La visibilité en dehors du package d'un symbole déclaré dans un package est contrôlée par l'utilisation d'une majuscule ou minuscule en premier caractère.
  - Une **majuscule** rendra le symbole publique et utilisable en dehors du package.
  - Une **minuscule** rendra le symbole privé seulement accessible dans le package courant.

```go
// fonction privée du package courant. Ne peut pas être utilisée a l'extérieur.
func privateFunc() {
        // appelle la fonction `readContent` du package os.
        // ❌ Ne compile pas: `readContent` n'est pas exportée.
        content := os.readContent("/some/file")
}

// fonction publique du package courant.
// Peut être appelée depuis un autre package.
func PublicFunc() {
        // appelle la fonction `OpenFile` du package os.
        // ✅ compile, `OpenFile` est exportée!
        file, _ := os.OpenFile("/some/file")
}
```

Copy

Contrôler la visibilité des symboles exporté permet de s’assurer que le package sera bien utilisé!

C’est l’idée **d’encapsulation**, on expose uniquement ce dont l’utilisateur à besoin.

Votre fichier peut dépendre d'un package issu de:

- Votre de projet courant
- La (très fournie) librairie standard de Go
- D'une librairie externe

```go
package main

import (
        // Imports de la librairie standard.
        "crypto/tls"
        "fmt"
        "io"
        "net/http"

        // Import du projet courant.
        "mondomaine.com/monprojet/pkg/helpers"

        // Imports de librairie externes (dépendances).
```

Copy

Variables (1/3)

Une variable est une zone mémoire allouée contenant une valeur

Copy

```go
func main() {
	// Déclare une variable de type string et assigne à la valeur par défaut du type.
	// "" pour string.
	var message string
	// Assigne (copie) la valeur "Hello ENSG!" a la variable message.
	message = "Hello ENSG!"

	// Est équivalent a:
	var message string = "Hello ENSG!"

	// Est encore équivalent a l'expression précédente avec une syntaxe compacte.
	// Ici le compilateur devine le type de la variable en fonction de la valeur assignée.
	message := "Hello ENSG!"

	// Affiche la valeur de la variable message dans la sortie standard.
	fmt.Println(message)
}
```

Variables (2/3)

Une variable est définie dans un "scope", par défaut une fonction. C’est sa durée de vie.

Copy

```go
func doSomething() {
        var age int64

        age = readAge()

        // ❌ Ne compile pas: newAge n'est pas définie dans ce scope.
        newAge = 45

        fmt.Println(age)
}

func readAge() int64 {
        // ❌ Ne compile pas. age n'est pas définie dans ce scope.
        age = 42

        newAge := readAgeFile()

        return newAge
}
```

Variables (3/3)

⚠ Go est un langage fortement typé ⚠

```go
func main() {
        // Déclare et initialise une variable message de type string.
        message := "Hello ENSG!"
        // Déclare et initialise une variable age de type int.
        age := 43

        // Assigne la valeur age dans la variable message.
        // ❌ Ne compile pas!
        // message: cannot use age (variable of type int) as string value in assignment.
        message = age
}
```

Copy

static type mental gymnastics
static types catch
errors at compile time
dynamic type mental gymnastics
you have to do REAL
dynamic typing like
Alan Kay
there are no
silver bullets
it's not a problem for
experienced developers
types slow
me down
test suite catches
errors anyway

Constantes

```go
const apiURL = "https://swapi.dev/vehicles/4"

func main() {
        const anotherConst = 4

        fmt.Println(apiURL, anotherConst)
}
```

Copy

Types Scalaires

- Numériques: `int`, `intX`, `uint`, `uintX`, `float32`, `float64`
- Booléen: `bool`
- Chaine de caractères UTF-8: `string`
- Autres: `byte` (octet), `rune` (caractère UTF-8)

Conversions

```
func main() {
        var i int = 42
        var f float64 = float64(i)
        var u uint = uint(f)
}
```

Copy

Controle de Flot

Controle de Flot: if (1/2)

Copy

```go
func main() {
	// if / else classique.
	ok := doSomething()
	if ok {
		fmt.Println("C'est OK!")
	} else {
		fmt.Println("C'est pas OK!")
	}
}

func doSomething() bool {
	return true
}
```

Controle de Flot: if (2/2)

Copy

```go
func main() {
	// if / else avec déclaration.
	// avantage: ok n'existe que dans le scope du if.
	if ok := doSomething(); ok {
		fmt.Println("C'est OK!")
	} else {
		fmt.Println("C'est pas OK!")
	}

	// Ne compile pas: ok n'est pas défini.
	ok = true
}

func doSomething() bool {
	return true
}
```

Controle de Flot: switch

Copy

```go
func main() {
        // switch
        age := readAge()
        switch age {
        case 10:
                fmt.Println("Hello 10")
        case 42:
                fmt.Println("Hello 42")
        default:
                fmt.Prinln("Hello darkness my old friend")
        }
}

func readAge() int {
        return 42
}
```

Controle de Flot: boucle for

```go
func sum0to9() {
	var total int

	for i := 0; i < 10; i++ {
		total += i
	}

	fmt.Println("Total", total)
}
```

Copy

Fonctions(1/4)

- Une fonction est un groupement logique d'instructions
- Accepte entre 0 et N arguments
- 🎉 Retourne entre 0 et N résultats 🎉

Copy

```go
// Un fonction qui accepte une string et ne retourne rien.
func sayHello(message string) {
	fmt.Println("Hello:", message)
}

// Une fonction qui accepte deux entiers et qui retourne un float64 et une erreur.
func divide(numerator, denominator int) (float64, error) {
	if denominator == 0 {
		return 0, errors.New("can't divide by 0")
	}

	return numerator / denominator, nil
}
```

Fonctions(2/4)

- Les fonctions peuvent être manipulées comme des valeurs!

Copy

```go
// Une fonction qui accepte une chaine de caractères
// ... et qui retourne une fonction qui n'accepte aucun argument
// mais qui retourne une chaine de caractères.
func messWithFuncs(name string) func() string {
	// Les fonctions peuvent etre manipulées comme des valeurs!
	fn := func() string {
		return "Hello " + name
	}

	return fn
}
```

- Go permets de "reporter" l'exécution d'une fonction quand une fonction parente se termine
- Pratique pour garantir qu'une ressource soit libérée quoi qu'il se passe lors de l'exécution de la fonction.
  - Similaire aux "destructeurs" en C++

```go
func faireDeLaPolitique() {
	rendreLargent := func() {
		fmt.Println("Argent rendu")
	}

	// Quoi qu'il advienne, l'argent sera rendu.
	// Peu importe le résultat des élections.
	defer rendreLargent()

	if elu := elections(); elu {
		fmt.Println("Je suis élu")
		return
	}
```

Copy

Fonctions(4/4)

- Les arguments de fonctions sont passés par valeur.
- Cela signifie que les valeurs des arguments sont copiés lors de l'appel

```go
func main() {
	name := "John"

	addGreeting(name)

	// Affiche "John" et non "Hello John".
	fmt.Println(name)
}

func addGreeting(name string) {
	name = "Hello " + name
}
```

Copy

9.33

Pointeurs (1/3)

- Déclarer une variable reviens à indiquer au programme d'allouer une certaine quantité de mémoire (en fonction du type de la variable) à une adresse en mémoire
- 🎓 Go permets de **référencer** cet emplacement mémoire en copiant son adresse dans une autre variable avec l'opérateur `&`. Autrement dit, on **crée un pointeur**.

Copy

```go
func main() {
        // On déclare et initialise une variable. Cela alloue de la mémoire sur la pile.
        var message string = "Hello ENSG!"
        // On copie l'adresse mémoire de cette variable dans une nouvelle variable.
        // Pour cela on utilise l'opérateur & (référence).
        var pointerToMessage *string = &message

        // Affiche: message address in memory is: 0xc000014070
        fmt.Println("message address in memory is:", pointerToMessage)
}
```

Pointeurs (2/3)

- À l'inverse, on peut aussi accéder au contenu d'une variable référencée par un pointeur.
- 🎓 Cela est appelé **déréférencer** un pointeur, avec l'opérateur `*`.

```
func main() {
        var message string = "Hello ENSG!"
        var pointerToMessage *string = &message

        // Affiche: message is: Hello ENSG!"
        fmt.Println("message is:", *pointerToMessage)
}
```

Copy

Pointeurs (3/3)

- 🎓 Les types **pointeur sur X** sont des types dit de **référence**.
- 🎓 La valeur par défaut d'un type référence est `nil`.
- Il existe d'autres types références en go.

```go
func main() {
        // Le `= nil` est optionnel ici: la valeur par défaut d'un pointeur est nil.
        var nilPointer *string = nil

        fmt.Println("address is:", nilPointer)
        // A votre avis: que fait cette ligne?
        fmt.Println("message is:", *nilPointer)
}
```

Copy

🎓 Exercice: Corriger la Fonction

- Corriger la fonction `addGreeting` pour qu'elle affiche correctement `Hello John`
- Sans retourner de valeur.

```go
func main() {
        name := "John"

        addGreeting(name)

        // Affiche "John" et non "Hello John".
        fmt.Println(name)
}

func addGreeting(name string) {
        name = "Hello " + name
}
```

Copy

✅ Solution: Corriger la Fonction

Copy

```go
func main() {
	name := "John"

	// On passe en argument de addGreeting l'adresse de la variable `name`.
	addGreeting(&name)

	fmt.Println(name)
}

func addGreeting(namePtr *string) {
	// La valeur de la variable référencée par namePtr égale à
	// la chaine de caractères "Hello " concaténée avec
	// la valeur de la variable référencée par namePtr.
	*namePtr = "Hello " + *namePtr
}
```

Gestion d'Erreur (1/2)

- Go traite les erreurs avec des valeurs retour au lieu d'exceptions
- Il est commun qu'une fonction qui puisse échouer retourne une valeur et un résultat.
  - Il convient alors de vérifier l'erreur retournée soit égale a nil.
  - …Sinon il faudra la gérer!

Copy

```go
func main() {
        file, err := os.Open("/super/file")
        if err != nil {
                // Si err est non nil, alors l'opération à échouée,
                fmt.Println("Impossible d'ouvrir le fichier", err)
                return
        }
        // On s'assure de toujours fermer le fichier ouvert.
        defer file.Close()

        // On peut intéragir avec le fichier!
}
```

Gestion d'Erreur (2/2)

- Certaines instructions peuvent mettre le programme dans un état ou il ne peut plus s'exécuter.
  - Par exemple, accéder à un pointeur `nil`
- Dans ce cas la, l'exécution de la fonction s'arrête et on parle de `panic`

Slices

- Une slice est un tableau dynamique de valeurs

Copy

```go
func main() {
        // Déclare une slice de string nil
        var slice []string

        // Déclare et initialise une slice d'entiers
        intSlice := []int{1, 2, 4, 5}

        // Ajoute une ou plusieurs valeurs a la même slice.
        intSlice = append(intSlice, 6, 7, 8)

        // Donne la taille d'une slice
        len(intSlice)
}
```

Parcourir une slice

- Go fournit la fonction `range` qui permet de parcourir une collection.
- `range` accepte une collection, et retourne deux valeurs:
  - L'index courant dans la collection
  - La valeur de la collection a l'index

```go
func main() {
	slice := []int{2, 4, 6, 8}

	// Affiche:
	// Index: 0 Value: 2
	// Index: 1 Value: 4
	// Index: 2 Value: 6
	// Index: 3 Value: 8
	for index, value := range slice {
		fmt.Println("Index: ", index, "Value: ", value)
	}
}
```

Copy

🎓 Exercice: Convertir une collection d'entiers en une collection de strings

- Ecrire une fonction `toStringSlice` qui convertit slice d'entiers en une slice de strings.
  - 🕵️: Il faut utiliser la fonction **strconv.Itoa** doc
  - Bonus: implémenter cette fonction sans utiliser `append`.

✅ Solution: Convertir une collection d'entiers en une collection de strings

Copy

```go
func main() {
	input := []int{1, 2, 3, 4}
	output := toStringSlice(input)
	fmt.Println(output)
}

func toStringSlice(input []int) []string {
	var result []string

	// Pour chaque entrée de la slice input...
	for _, v := range input {
		// On ajoute la slice de résultat
		result = append(result, strconv.Itoa(v))
	}

	return result
}
```

- Type déclaré représentant une collection fixe d'attributs (aussi appelés membres)
- Les attributs commençant par une lettre majuscules sont accessibles en dehors du package. Ceux qui commencent par une lettre minuscule ne le sont pas.

Copy

```go
// Déclaration du type lecture, composé de 3 attributs.
type Lecture struct {
	Topic    string
	Duration time.Duration
	Credits  int
}

func main() {
	// On declare et initialise une nouvelle variable de type Lecture.
	coursCICD := Lecture{
		Topic:    "CICD",
		Duration: 3 * 6 * time.Hour,
		Credits:  2,
	}
	// On prends la référence de la variableCICD
	var ptrVersCoursCICD *Lecture = &coursCICD

	// On accède aux valeurs des membres de la variable coursCICD avec `.`
```

Structures (2/3)

- La valeur par défaut d'une structure est égale a l'ensemble des valeurs par défaut de ses membres.

Copy

```go
// Déclaration du type lecture, composé de 4 attributs.
type Lecture struct {
	Topic    string
	Duration time.Duration
	Credits  int
	// attribut secret, seulement accessible dans le package courant.
	secret string
}

func main() {
	coursVide := Lecture{
		Topic:    "",
		Duration: time.Duration(0),
		Credits:  0,
	}
	coursDéfaut := Lecture{}

	if coursVide == coursDéfaut {
		// Affiche OK.
		fmt.Println("OK")
	}
```

9.46

- Toute structure doit instanciée doit être dans un état utilisable.
- Si les valeurs par défaut ne suffisent pas, on peut fournir une fonction d'initialisation.
  - Similaire aux "constructeurs" dans d'autres langages.

Copy

```go
// Déclaration du type lecture, composé de 4 attributs.
type FileReader struct {
	File *os.File
}

func NewFileReader(path string) (*FileReader, error) {
	file, err := os.Open(path)
	if err != nil {
		return nil, err
	}

	return &FileReader{
		File: file,
	}
}

func main() {
```

Annotations de Structures

- Go permets d'annoter les membres d'une structure
- Utilisation diverses
  - Ex: indiquer le champ d'un objet JSON a lié à l'attribut.

```
type City struct {
        ID   string `json:"id"`
        Name string `json:"name"`
}

// {"id": "1", "name": "Lyon"}
```

Copy

Définir des types

- Nous avons utilisé le mot clef `type` pour définir un nouveau type de structure.
- Mais `type` peut être appliqué à tous les types existants.

```go
// Déclare un type Color représenté par un entier.
type Color int

const (
        ColorBlue  = 0
        ColorGreen = 1
)

// Déclare un type Car représenté par une structure composé de trois attributs.
type Car struct {
        Color   Color
        Engine  Engine
        Battery Battery
}

// Déclare un type Garage, qui est une map entre le nom du propriétaire et la voiture.
type Garage map[string]Car
```

Copy

9.49

- Il est possible d'attacher des méthodes aux types que l'on définit en utilisant la syntaxe suivante
- Le premier argument avant le nom est appelé **receveur**, c'est la référence vers l'objet sur lequel la méthode est appelée

Copy

```go
// Définit un type Color représenté en mémoire par un entier.
type Color int

// Attache une méthode String a toute instance de la valeur Color
// qui retourne le nom de la couleur sous forme de chaine de Caractères.
func (c Color) String() string {
	switch c {
	case ColorBlue:
		return "blue"
	case ColorGreen:
		return "green"
	default:
		return "unknown"
	}
}

// Bloc de constantes déclarant les couleurs possibles
const (
```

Types et Méthodes (2/2)

- Une structure avec des méthodes est l'équivalent d'une classe dans d'autres langages.
- Le "Receveur" est équivalent a **this** en C++ ou Java.

Copy

```go
type Car struct {
        Brand string
        Color Color
}

// Attache une méthode a toute instance de type "pointeur sur Car".
// Le premier argument avant le nom de la méthode est appelé "receveur".
func (c *Car) Describe() {
        fmt.Printf("Car brand is: %s, car color is %s\n", c.Brand, c.Color.String())
}

func main() {
        car := Car{
                Brand: "Renault",
                Color: ColorBlue,
        }

        car.Describe()
```

Receveurs: Pointeurs ou Valeurs?

- On peut attacher une méthode sur une valeur du type, ou sur un pointeur.
- L'opérateur `.` (accés) référence et déréférence les pointeurs implicitement.
- On utilise une valeur quand on souhaite que la méthode ne puisse pas changer l'objet sur laquelle elle est appelée
- ⚠ Utiliser une valeur fait une copie à chaque appel.

Types Abstraits: Interfaces (1/3)

- Une interface décrit un jeu de méthodes.
- Une variable du type de l'interface peut recevoir n'importe quel type qui implémente les méthodes de l'interface.
- Le comportement d'un appel de méthode est celui du type concret caché derrière l'interface. C'est ce qu'on appelle le **Polymorphisme**.

Copy

```go
type Vehicle interface {
	Ride()
}

type Scooter struct{}

func (s *Scooter) Ride() {
	fmt.Println("Riding a Scooter")
}

type Bicycle struct{}

func (b *Bicycle) Ride() {
	fmt.Println("Ride a Bicycle")
}

func main() {
	// La variable vehicle peut recevoir soit un Scooter, soit un Bicycle.
	// Ces deux types satisfont l'interface `Vehicle`.
	var vehicle Vehicle

	vehicle = &Scooter{}
	// Affiche "Riding a Scooter".
	vehicle.Ride()

	vehicle = &Bicycle{}
	// Affiche "Riding a Bicycle".
```

9.54

Types Abstraits: Interfaces (3/4)

- Une interface est un type référence vers un autre type, sa valeur par défaut est `nil`.
- Les interfaces sont implicites:
  - Du moment que le type de la valeur satisfait toutes les méthodes de l'interface, alors il est considéré comme implémentant l'interface.
  - Pas de mot clé `implements` comme en Java

- Pourquoi s'embêter à faire des interfaces?
  - Fournir du code générique
  - Découpler, cacher la complexité
    - Exemple: Cacher une dépendance à une base de données derrière une interface

```go
// writeHello écrit hello dans n'importe quelle destination du moment qu'elle satisfait `io.Writer`
func writeHello(dest io.Writer) {
	dest.Write([]byte("hello"))
}

func main() {
	var buf bytes.Buffer

	// Ici on écrit dans un buffer en mémoire.
	writeHello(&buf)

	file, _ := os.Open("./file")
	defer file.Close()
```

Copy

9.56

Interfaces Importantes en Go

- package `io`:
  - io.Reader, io.Writer, io.Closer
- package `http`:
  - http.Handler permet de gérer et répondre à une requête HTTP.
- Le type `error`

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

Copy

9.57

Nommage de variables

- **Convention**: La longueur du nom d'une variable est proportionnelle à sa durée de vie.
- Certaines exceptions:
  - `err` pour une valeur d'erreur
  - `ctx` toute instance d'un a `context.Context`
  - `r` et `rw` pour requête et response writer dans un handler HTTP.
  - Receveurs de méthodes (ie `*Store` ⇒ `st`)
  - Et d'autres…

Quelques Exemples

- Décoder du JSON vers une structure

Copy

```go
type City struct {
        ID   string `json:"id"`
        Name string `json:"name"`
}

// {"id": "1", "name": "Lyon"}


const payload = `[{"id":"1","name":"Lyon"},{"id":"2","name":"Paris"}]`

func main() {
        var cities []City

        if err := json.Unmarshal([]byte(payload), &cities); err != nil {
                fmt.Println("cannot unmarshal", err)
                return
        }

        fmt.Println(cities)
}
```

Quelques Exemples

- Lire l'intégralité d'un Buffer

```go
func main() {
	var buf bytes.Buffer

	readBytes, err := io.ReadAll(&buf)
	if err != nil {
		// KO.
	}
	// OK!
}
```

Copy

Quelques Exemples

- Faire une requête HTTP

Copy

```go
func main() {
	resp, err := http.Get("https://google.com")
	if err != nil {
		// Handle err...
	}
	if resp.StatusCode != http.StatusOK {
		// Code non OK. Ouch.
	}

	// On va accéder au Body.
	// Il faut s'assurer de toujours le fermer.
	defer resp.Body.Close()
}
```

🎓 Exercice: Donnez le Climat de Tatooine

On vous demande le programme go suivant:

- Fait une requête a swapi.dev pour récupérer les informations de la planete Tatooine
  - L'URL a appeler est https://swapi.dev/api/planets/1/
- Si la requête est réussie alors on lit l'intégralité du corps de la réponse
- Et on déserialise la réponse dans un objet pour accéder au `climat`
- **Bonus**: Une fois fini, essayez de nous donner le diametre de l'étoile noire aussi.

Copy

```go
type Planet struct {
	Climate string `json:"climate"`
}

func main() {
	resp, err := http.Get("https://swapi.dev/api/planets/1")
	if err != nil {
		fmt.Println("Cannot query swapi", err)
		return
	}

	if resp.StatusCode != http.StatusOK {
		fmt.Println("Bad status", resp.Status)
		return
	}

	defer resp.Body.Close()

	readBody, err := io.ReadAll(resp.Body)
	if err != nil {
		fmt.Println("Can't read body", err)
		return
	}

	var tatooine Planet
	if err := json.Unmarshal(readBody, &tatooine); err != nil {
		fmt.Println("Can't unmarshal payload", err)
```

9.63

- Vidéos
  - https://www.youtube.com/watch?v=xi8732QO33Y
  - https://www.youtube.com/c/JustForFunc
- Guides complets
  - https://go.dev/tour/
  - https://gobyexample.com/
- Pour aller plus loin
  - https://go.dev/doc/effective_go
  - https://go.dev/doc/

Bonus Track

Collections: Arrays

- Un tableau de taille fixe de N elements.
- ⚠ La taille du tableau fait partie de son type
  - **Limite**: ne peut pas être changée une fois le tableau instancié.

Copy

```go
func main() {
        // Declare et initialise un tableau de 2 entiers.
        var intArray [2]int
        // On peut assigner un élément du tableau en utilisant son index.
        intArray[0] = 1
        intArray[1] = 3
        // On accède a un élément du tableau en utilisant son index.
        fmt.Println(intArray[0], intArray[1])

        anotherArray := [4]int{2, 4, 6, 8}
        // ❌ Ne compile pas: la taille fait partie du type!
        // On assigne un tableau de 4 entrées a un tableau de deux entrées
        intArray = anotherArray
}
```

Collections: Slices (1/5)

Une `slice` est une référence sur un sous ensemble d'entrées dans un tableau

Copy

```
func main() {
        anArray := [4]int{2, 4, 6, 8}

        // Déclare et initialise une slice référençant les entrées
        // entre l'index 1 et 3 du tableau anArray.
        // Se lit intervalle [1:4[, du coup 1,2 et 3.
        var aSlice []int = anArray[1:4]

        // ⚠️ Une écriture écrit une valeur dans le tableau référencé!
        aSlice[0] = 9
        fmt.Println(aSlice)  // [9, 6, 8]
        fmt.Println(anArray) // [2, 9, 6, 8]
}
```

Collections: Slices (2/5)

- On peut initialiser directement une slice sans passer par un tableau.
- On peut aussi initialiser une slice avec l'opérateur `make`

```go
func main() {
	aSlice := []int{2, 4, 6, 8}
	// Sélectionne les entrées entre l'index 2 et 3 de la slice aSlice.
	anotherSlice := aSlice[2:4]
	fmt.Println(aSlice)       // [2, 4, 6, 8]
	fmt.Println(anotherSlice) // [6, 8]

	// Initialise une slice de strings de 3 entrées.
	yetAnotherSlice := make([]string, 3)
	fmt.Println(yetAnotherSlice) // ["", "", ""]
}
```

Copy

Collections: Slices (3/5)

Une `slice` possède deux caractéristiques importantes:

- Sa taille: le nombre d'éléments présents dans la slice
  - On y accède a l'aide de la fonction `len`
- Sa capacité: la taille totale du tableau référencé
  - On y accède a l'aide de la fonction `cap`

```go
func main() {
        sliceOne := []int{0, 1, 2, 3}
        sliceTwo := sliceOne[0:2]
        // Affiche "Length: 2 Capacity: 4"
        fmt.Println("Length: ", len(sliceTwo), "Capacity: ", cap(sliceTwo))
}
```

Copy

?

9.69

- On peut concaténer des objets à une slice avec l'opérateur `append`
- ⚠ Cela n'ajoute pas nécessairement un entrée a la slice passée en parametre.
  - Dans le cas ou le tableau sous-jacent est plein (len == cap), `append` va réallouer un tableau et copier toutes les entées dans ce nouveau tableau.
  - 👉 En conséquence: il faut **TOUJOURS** assigner la valeur retournée par `append`

```go
func main() {
        // On ajoute l'entrée 10 a la slice aSlice
        aSlice := []int{2, 4, 6, 8}
        aSlice = append(aSlice, 10)
        fmt.Println(aSlice) // [2, 4, 6, 8, 10]

        // On ajoute tous les items de la `anotherSlice` a la slice `aSlice`
        // Et on assigne le résultat à la variable yetAnotherSlice
        // Notez les  "..."
        anotherSlice := []int{10, 12, 14, 16}
        yetAnotherSlice := append(aSlice, anotherSlice...)
```

Copy

Collections: Slices (5/5)

- Le type **slice de X**, comme le type **pointeur sur X**, est un type référence.
- Sa valeur par défaut est `nil`
- Accéder à une slice nil provoque une panic
- En revanche: `append` et `len` savent gérer une nil slice.

```go
func main() {
	var nilSlice []string

	// panic!: on accède a un tableau qui n'existe pas.
	v := nilSlice[0] // 💥

	fmt.Println(len(nilSlice), cap(nilSlice)) // 0, 0

	nilSlice = append(nilSlice, "foo", "bar", "biz")
	fmt.Println(nilSlice) // ["foo", "bar", "biz"]
}
```

Copy

Collections: maps (1/3)

- Tableau associatif (clé → valeur)
- Initialisée de façon littérale, ou avec `make`
- On récupère sa taille avec `len`
- On supprime une clé avec `delete`
- Type référence, comme les pointeurs out les slices
  - Une map peut être nil, `len` retournera 0.

Collections: maps (2/3)

Exemple d'écriture

Copy

```go
func main() {
	// Déclaration et initialisation d'une map de façon littérale.
	mapAges := map[string]int{
		"Julien": 35,
		"Damien": 36,
	}

	// Déclaration et initialisation d'une map de taille 2.
	mapVilles := make(map[string]string, 2)
	// Ecritures des valeurs dans la map.
	mapVilles["Julien"] = "Lyon"
	mapVilles["Damien"] = "St-Etienne"

	var nilMap map[int]int
	nilMap[21] = 42 // panic! écriture dans une map qui n'est pas instanciée

	// On peut suprimer une entrée d'une map
	delete(mapVilles, "Julien")

	// Affiche 2, 1, 0.
	fmt.Println(len(mapAges), len(mapVilles), len(nilMap))
}
```

Collections: maps (3/3)

Exemple de lecture

Copy

```go
func main() {
	// Déclaration et initialisation d'une map de façon littérale.
	mapAges := map[string]int{
		"Julien": 35,
		"Damien": 36,
	}

	// Lecture sans vérification.
	// Si la clé existe, retourne la valeur associée.
	// Si la clé n'existe pas, retoure la valeur par défaut du type de la valeur.
	ageJulien := mapAges["Julien"]

	fmt.Println("Age de Julien", ageJulien)

	// Lecture avec vérification.
	// Si la clé existe, la valeur sera retournée, et ok sera a true
	// Si la clé n'existe pas, ok sera false.
	ageMichel, ok := mapAges["Michel"]
	if !ok {
		fmt.Println("Pas d'age pour Michel")
	} else {
		fmt.Println("Age de Michel", ageMichel)
	}
```

Parcourir une map

- `range` supporte aussi les maps dans une boucle for
- Assigne la clé et la valeur courante
- ⚠ L'ordre de parcours n'est pas déterministe! Il ne faut pas en dépendre!

```go
func main() {
	mapAges := map[string]int{
		"Julien": 35,
		"Damien": 36,
	}

	// Affiche soit:
	// Julien a 35 ans
	// Damien a 35 ans
	// OU
	// Damien a 35 ans
	// Julien a 35 ans
	for name, age := range mapAges {
		fmt.Printf("%s a %d ans\n", name, age)
	}
```

Copy

9.75

🎓 Exercice: Comptez les occurrences de mots dans une chaine de caractère

- Écrivez une fonction `WordCount` en go qui accepte une chaine de caractère et qui retourne le nombre d'occurrences de chacun des mots contenu dans la chaine.
  - **Indice**: La signature de votre fonction devrait ressembler a `func WordCount(str string) map[string]int`.
    - La valeur de retour mappe le mot vers le nombre de fois qu'il est apparu.
  - **Indice**: `strings.Fields` (doc) sépare les mots d'une chaine de caractère et retourne une string.

✅ Solution: Comptez les occurrences de mots dans une chaine de caractère

Copy

```go
func main() {
	input := "The quick quick brown fox jumps over the lazy lazy dog"

	result := WordCount(input)

	fmt.Println(result)
}

func WordCount(input string) map[string]int {
	result := make(map[string]int)

	for _, word := range strings.Fields(input) {
		result[word]++
	}

	return result
}
```

Présentation de votre projet

Contexte(1/4)

- Voi est une société qui fournit un service de "véhicules de transport doux" à la demande
  - 🔓 Vous déverrouillez un véhicule avec votre Smartphone
  - 🛴 Vous faites votre trajet avec le véhicule
  - 🔒 Vous verrouillez et laissez le véhicule sur votre lieu d’arrivée
  - 💸 Vous payez le temps passé sur le véhicule

Contexte(2/4)

Un cas d'utilisation majeur est de permettre aux utilisateurs de trouver un véhicule proche d'eux facilement.

Contexte(3/4)

- Ce service est assuré par une API HTTP appelée `vehicle-server` qui doit supporter les fonctionnalités suivantes:
  - Enregistrer un véhicule
  - Trouver les N véhicules les plus proches de soi
  - Supprimer un véhicule

Contexte (4/4)

- Voi est entrain de reconstruire cette fonctionnalité et à décidé de sous-traiter le développement de ce projet a l'ENSG…
- Une équipe technique de **Voi** avait commencé l'implémentation du serveur, et vous à mis à disposition une archive téléchargeable ICI, contenant le code source du projet.

Prise en Main du Projet (2/2)

```
# Création du répertoire vehicle-server
mkdir -p /workspace/vehicle-server && cd /workspace/vehicle-server

# Téléchargez le projet sur votre environnement de développement
curl -sSLO https://cicd-lectures.github.io/slides/main/media/vehicle-server.tar.gz

# Décompresser et extraire l'archive téléchargée
tar xvzf ./vehicle-server.tar.gz
```

Copy

A partir de la vous pouvez ouvrir le fichier `README.md` et commencer à suivre ses instructions.

Qu’est-ce qui va / ne va pas dans ce projet d’après vous?

Triste Rencontre avec la Réalité

- Pas de gestion de version
- Le projet ne fonctionne pas complètement, delete réponds un erreur 😭
- Il suffirait de l'implémenter
- … sauf que vous ne pouvez pas compiler le projet!

🎓 Exercice : Initialisez un dépôt git

- Supprimez le binaire et l'archive et initialisez un dépôt git dans le répertoire, puis créez un premier commit

✅ Solution Exercice

```
rm -f server vehicle-server.tar.gz

# On initialise un nouveau dépôt git
git init

# On ajoute tous les fichiers contenus a la zone de staging.
git add .

# On crée un nouveau commit
git commit -m "Add initial vehicle-server project files"
```

Copy

Checkpoint 🎯

- Vous avez récupéré un projet Go qui semble fonctionner…
  - ..mais pas vraiment à l'état de l'art.
- Application du chapitre précédent : vous avez initialisé un projet `git` local

Cycle de vie de votre projet

🤔 Quel est le problème ?

On a du code. C’est un bon début. MAIS:

- Qu’est ce qu’on "fabrique" à partir du code ?
- Comment faire pour "fabriquer" de la même manière pour tout•e•s (💻 | 🖥️) ?

Que "fabrique" t’on à partir du code ? 📦

Un **livrable** :

- C’est ce que vos utilisateurs vont utiliser: un binaire à télécharger ? L’application de production ?
- C’est versionné
- C’est *reproductible*

Que signifie "reproductible" ?

- Il faut que notre processus de génération de livrable, (notre build) soit entièrement **déterministe**.
- Il faut qu’en fonction d’un jeu de paramètres, le résultat du processus de livraison soit même le "même".
- Il en va de même pour l’environnement de production

Quels sont les paramètres de notre livraison ?

- **Le code**: Dans quelle version est-il? Est-il fonctionnel? Est-ce qu'il est sauvegardé?
- **Les dépendances de notre code**: Toutes les libraires utilisés dans notre application.
- **Les outils de génération de livrables**: Quel compilateur et dans quelle version?
- **L'environnement d'exécution cible**: Java 17 ou Java 20? Quelle version de PostgreSQL? Quel OS/Architecture CPU? Quel Navigateur?
- **Le processus de livraison lui même**: Dans quelle mesure la procédure de génération est elle répétable et respectée?

Risques encourus?

- 😡 Dans le meilleur des cas, votre livrable ne marche pas du tout.
- 🤡 Dans certains cas votre livrable va casser sans explication facile et seulement sur la production du client les jours impairs d'une année bisextile.
  - Allez reproduire et débugger!
- 😱 Livrer votre application va devenir une angoisse permanente
- 😱😭🔥☠️ Vous livrez une CVE ou un malware, avec un accès direct a votre base de données.
  - Vraiment
  - Vraiment, Vraiment
  - Vraiment, Vraiment, Vraiment…

11.7

On en est où la dedans? (1/2)

- **Le code**
  - ✅ On vient de mettre en place git. On sait identifier une version par un hash de commit.
  - ❌ On ne sait pas vraiment dire si l'application "fonctionne" ou pas.
- **Les dépendances de notre code**:
  - ❌ On ne sait ni les récupérer, ni les contrôler.
- **Les outils de génération de livrables**
  - ❌ On sait que go1.22 est indiqué dans la documentation fournie mais c'est tout.

On en est où la dedans? (2/2)

- **L'environnement cible**:
  - ⚠ La compilation Go génère un binaire qui embarque son environnement d'exécution. C'est donc lié à la version du compilateur Go. Ce n'est pas le cas pour d'autres langages.
  - ✖ Par contre on sait que l'on à besoin de Postgres et Postgis, mais pas grand chose de plus!
  - ✓ Voi nous demande de cibler Linux >= 5.x sur une architecture CPU amd64
- **Le processus de livraison lui même**:
  - ✖ Nous n'avons encore rien défini

Quelles solutions ? (1/2)

- **Le code**
  - ➡ **Solution** (pour savoir si il fonctionne): **les tests automatisés**
  - ➡ **Solution** (pour garantir qu'il fonctionne à chaque changement): **l'intégration continue (CI)**
- **Les dépendances du code**
  - ➡ Solution: Mise en place d'outils de **gestion et d'audit des dépendances**
- **Les outils de génération du code**:
  - ➡ Solution: Mise en place d'un environnement contrôlé et automatisé de génération de livrable, via de la **Livraison Continue**

Quelles solutions ? (2/2)

- **L'environnement cible**:
  - ➡ Solution: Utilisation **d'outils de packaging** (Docker) pour notre application et son environment cible
- **Le processus de livraison lui même**:
  - ➡ Solution: définir un **cycle de vie** et en déduire un **processus de livraison**

Le cycle de vie de notre application

- `build`: Compilation de l'application
- `lint`: Analyse statique de code pour détecter des problèmes ou risques
- `test`:
  - `unit_test`: Exécution de tests unitaires
  - `integration_test`: Exécution des test d'intégration
- `package`: Création du livrable
- `release`: Livraison du livrable

Comment normaliser ce cycle de vie?

- Tout le monde peut jouer des commandes comme il le souhaite
- Il est nécessaire que tous les acteurs (développeurs et CI) jouent les même commandes
- Utilisation d'un outil `normaliser` ces commandes
  - ➡ On propose se propose d'utiliser `make`

`make`, kesako?

- GNU Make est un outil en ligne de commande,
- qui lit un fichier `Makefile` pour exécuter des tâches.
- Chaque tâche (ou "règle") est décrite par une "cible":
- Format d'une "cible" make :

```
cible: dependance
        commandes
```

Copy

- On appelle la commande `make` avec une ou plusieurs cibles en argument :

```
make clean build
```

Copy

Exemple de Makefile

```
# Fabrique le fichier "hello" (binaire) à partir des fichier "hello.o" et "main.o"
hello: hello.o main.o
        gcc -o hello hello.o main.o

# Fabrique le fichier "hello.o" à partir du code source "hello.c"
hello.o: hello.c
        gcc -o hello.o -c hello.c

# Fabrique le fichier "main.o" à partir du code source "main.c"
main.o: main.c
        gcc -o main.o -c main.c
```

Copy

```
make hello # Appelle implicitement "make hello.o" et "make main.o"
## équivalent à "make hello.o main.o hello"
```

Copy

🎓 Exercice: Mettre en place un Makefile dans le Projet

- La compilation doit générer le binaire dans le répertoire `dist`.
- On souhaite mettre en place un `Makefile` qui définit les cibles suivantes:
  - `dist`: crée le répertoire
  - `clean`: supprime le répertoire
  - `all`: qui exécute `clean` puis `dist`

✅ Solution: Mettre en place un Makefile dans le Projet

```
cd /workspace/vehicle-server
touch Makefile
```

Copy

```
all: clean dist

clean:
  rm -rf ./dist

dist:
  mkdir dist
```

Copy

Makefile Avancé (1/2)

- Par défaut une cible/règle correspond à un fichier
  - Si le fichier existe, `make` ne ré-exécutera pas les commandes
  - 🤔 Que se passe t'il si vous créez un fichier `all` dans le même répertoire que le Makefile?

Makefile Avancé (2/2)

- Pour désactiver ce comportement pour une cible donnée, ajoutez ladite cible comme dépendance à la cible spéciale `.PHONY`
  - On peut répéter `.PHONY` plusieurs fois
  - **convention**: on ajoute la cible à .PHONY avant sa définition

```
.PHONY: target
target: dependence
  commande
```

Copy

- Si vous appelez `make` sans argument, alors la cible par défaut sera la première cible définie

🎓 Exercice: Ajouter build dans le Makefile

- Mettez à jour votre Makefile pour introduire une cible `build`
- `build` doit générer le binaire dans le répertoire `./dist`
  - **Indice**: Le package main de notre serveur se trouve dans `./cmd/server`
  - **Indice**: `go help build`
- `build` doit être inclus dans la cible `all`
- `build` doit s'exécuter même si un fichier `build` existe

✅ Solution: Ajouter build dans le Makefile

Copy

```
.PHONY: all
all: clean dist build

.PHONY: clean
clean:
  rm -rf ./dist

.PHONY: build
build:
  go build -o ./dist/server ./cmd/server

dist:
  mkdir ./dist
```

😱 ça ne compile pas!

```
go: cannot find main module, but found .git/config in /workspace/vehicle-server
        to create a module there, run:
        go mod init
```

Copy

On ne peut pas compiler sans avoir auparavant réglé la question des dépendances!

Quelques cibles à rajouter :

Copy

```
DB_CONTAINER_NAME=vehicle-server-dev
POSTGRES_USER=vehicle-server
POSTGRES_PASSWORD=secret
POSTGRES_DB=vehicle-server
DATABASE_URL=postgres://$(POSTGRES_USER):$(POSTGRES_PASSWORD)@localhost:5432/$(POSTGRES_DB)

.PHONY: dev
dev:
	go run ./cmd/server \
		-listen-address=:8080 \
		-database-url=$(DATABASE_URL)

.PHONY: dev_db
dev_db:
  docker container run \
		--detach \
		--rm \
		--name=$(DB_CONTAINER_NAME) \
		--env=POSTGRES_PASSWORD=$(POSTGRES_PASSWORD) \
		--env=POSTGRES_USER=$(POSTGRES_USER) \
		--env=POSTGRES_DB=$(POSTGRES_DB) \
		--publish 5432:5432 \
		postgis/postgis:16-3.4-alpine

.PHONY: stop_dev_db
stop_dev_db:
```

Checkpoint 🎯

On a vu dans ce chapitre:

- Ce qu'est la reproductibilité des livrables et son importance
- On à défini un cycle de vie pour notre application
- On à découvert l'outil `make` pour implémenter ce cycle de vie

✅ On vient de terminer un chapitre, faites donc un commit!

La Gestion des Dépendances

🇺🇸 Dependency Management

Dans ce chapitre on ne parle que de dépendances du code

Pourquoi réutiliser du code?

- 🧱 L'informatique moderne est un assemblage de briques logicielles
- ⚙️ … chacune des briques étant infiniment complexe
  - Ex: TLS, PostgresSQL, Linux, Firefox…
- 🤔 Il est difficilement envisageable de démarrer un projet sans réutiliser des briques logicielles.
- 🧘 Cela permet de **concentrer son effort de développement sur ce qui apporte vraiment de la valeur**.
  - ➡️ Dans notre cas, notre métier est la gestion de véhicules, pas l'implémentation d'une pile réseau et d'un serveur HTTP.

⚠️ Ajouter une dépendance n'est pas un acte anodin ⚠️

- Si votre dépendance ne fonctionne plus ou est compromise, votre livrable sera impactée
- Attention à ne pas rajouter une dépendance trop grosse pour n'utiliser qu'une petite fonctionnalité!
- Attention aux dépendances de vos dépendances 😱
- Quelques règles d'usage:
  - Vérifier que votre dépendance est activement maintenue? (date du dernier commit, existence d'une communauté autour)
  - 👀 le code. Est-ce que vous le comprenez? Est-ce que vous pourriez le debugger ou le faire vous même?

Dépendre de librairies externes pose une quantité de problèmes!

- Comment récupérer tout le code dont on à besoin?
- Comment le maintenir à jour?
- Comment s’assurer qu’il n’a pas été modifié?
- Comment garantir la reproductibilité de nos builds?

What you think your dependency graph looks like

What your dependency graph actually looks like

12.6

BABEL-RUNTIME
babel-types
babel-runtime
babel-traverse
BUILDMAIL
ES5-EXT
express
lodash
babel-core
debug
grunt
inquirer
MONGOOSE
request
yeoman-generator
YEOMAN-GENERATOR
chalk
source-map
minimist
mkdirp
jsdom
broccoli-persistent-filter
node-gyp
glob
NPM
npm
graceful-fs
minimatch

12.7

12.8

Un peu de terminologie

- Une **dépendance** est une librairie de code externe qui fournit une fonctionnalité.
- On distingue deux types de dépendances:
  - Une **dépendances directe**: référencée directement par notre application
  - Une **dépendances transitive**: référencée par une des librairies dont l'application dépends

Comment résoudre le problème?

- On introduit un outil de gestion de dépendances
  - Permets au développeur de définir une liste de dépendances en fixant ou en plaçant une contrainte de version (ex $\Leftarrow$ 4.3.0)
  - Construit un graph de dépendances et le résous de façon a obtenir une liste de dépendances à télécharger de façon déterministe
  - Télécharge les dépendances sur la machine a distance et les mets à disposition de l'application.

Comment cela fonctionne avec go?

source

Quelques rappels sur go

- Tout symbole go est défini dans un package
- Un package est identifié par une URL unique
  - (ex: `github.com/prometheus/client_golang`)
- Un package peut importer un ou plusieurs autres packages pour utiliser ses symboles.

Provenances des packages

- La librairie standard
  - ✅ La version est contrôlée par la version de go utilisée pour compiler
- La base de code du projet
  - ✅ Contrôlée par notre VCS
- D'une ou plusieurs librairies externes
  - ❌ Pas de solutions pour l'instant

Bonjour go modules

- Go introduit la notion de **modules**: une collection de packages qui sont livrés, versionnés et distribués ensemble
- Un module est identifié par un **module path** déclaré dans un fichier appelé `go.mod`
- Le fichier `go.mod` est situé a la racine du module et englobe tous les packages définis en dessous de ce fichier

Le module github.com/prometheus/client_golang

```
├── go.mod # Racine du module github.com/prometheus/client_golang
├── go.sum
├── api # Package github.com/prometheus/client_golang/api
│   ├── client.go
│   ├── client_test.go
│   └── prometheus
│       └── v1 # Package github.com/prometheus/client_golang/api/prometheus/v1
│           ├── api.go
│           ├── api_bench_test.go
│           ├── api_test.go
│           └── example_test.go
├── prometheus # Package github.com/prometheus/client_golang/prometheus
│   ├── gauge.go
│   ├── gauge_test.go
├ [...]
[...]
```

Copy

Que déclare le fichier go.mod

- Le module path: l'identifiant unique du module
- La version minimum de go nécessaire pour utiliser ce module
- Les modules dont dépends le module courant
  - ⚠ La version **minimale** à utiliser est indiquée explicitement

Copy

```go
// Module path
module github.com/jlevesy/prometheus-elector

// Minimum go version
go 1.21

// Dépendances directes
require (
        github.com/fsnotify/fsnotify v1.6.0
        github.com/imdario/mergo v0.3.16
        github.com/prometheus/client_golang v1.17.0
        github.com/stretchr/testify v1.8.4
        golang.org/x/net v0.15.0
        golang.org/x/sync v0.3.0
        gopkg.in/yaml.v2 v2.4.0
        k8s.io/apimachinery v0.28.2
        k8s.io/client-go v0.28.2
        k8s.io/klog/v2 v2.100.1
)

// Dépendances contraintes indirectement (...)
// Certaines librairies ne supportent pas go modules, ils sont gérés comme
// dépendances directes...
// D'autres cas peuvent mener à l'ajout d'une dépendance indirecte.
require (
        github.com/beorn7/perks v1.0.1 // indirect
        github.com/cespare/xxhash/v2 v2.2.0 // indirect
```

🎓 Exercice: Ajout de la gestion de dépendances (1/3)

- On initialise un nouveau module

```
go mod init github.com/${VOTRE_UTILISATEUR_GITHUB}/vehicle-server
```

Copy

- On ajoute les dépendances au projet

```
go mod tidy
```

Copy

- ✕ Cela ne fonctionne pas, que se passe t'il?

🎓 Exercice: Ajout de la gestion de dépendances (2/3)

- Les fichiers go du projet importent encore des packages issus du module original!
  - `github.com/cicd-lectures/vehicle-server`
- Il faut les renommer, pour cela utilisez la commande suivante!

```
find . -type f -name '*.go' -exec sed -i -e 's,github.com/cicd-lectures,github.com/${VOTRE_UTILISATEUR_GITHUB},g' {} \;
```

Copy

- Pour vérifier que cette (horrible) commande a fonctionné on peut utiliser `git diff`
- Une fois cela fait on peut relancer

Copy

🎓 Exercice: Ajout de la gestion de dépendances (3/3)

- Affichez maintenant le graph de dépendances du projet

```
go mod graph

# Regarder uniquement les dépendances de pgx
go mod graph | grep pgx
```

Copy

- La partie droite indique le module parent, la partie gauche les modules dont dépend le module parent.
- ⚠ C'est verbeux, vous pouvez utiliser `| grep` pour filtrer le résultat
- Et pour finir, on oublie pas de créer un commit :)

Résolution Reproductible: l'algorithme MVS (1/2)

- **MVS**: Minimum Version Selection
- Go cherche à utiliser la **version minimale** d'une dépendance
- L'algorithme **est déterministe**:
  - Il ne change pas si une nouvelle version d'un module est mise à disposition.
  - Pas besoin de `lockfile` (comme sur npm, bundler, pip etc…)

Résolution Reproductible: l'algorithme MVS (2/2)

Contrôler le Contenu Téléchargé

- `go` lit le fichier `go.mod` du module à compiler et détermine la liste des modules à télécharger
- Il les télécharge soit depuis:
  - Un `module proxy`
  - Depuis un dépôt de code directement (via git, mercurial etc…)
- `go` calcule ensuite une somme de contrôle (checksum) des fichiers téléchargés et la compare avec un référentiel établi lors du premier téléchargement: le fichier `go.sum`.
  - ➡ Cela garantit que les dépendances téléchargées n'ont pas été altérées entre deux téléchargements.

Quelques Commandes Utiles:

- `go get <modulepath>@version`: Ajoute un module à la liste de dépendances
- `go mod tidy`: Nettoie le fichier `go.mod` en récupérant tout les modules importés par l'application et en s'assurant que le fichier `go.sum` est à jour.
- `go get -u ./...` Mets à jour tous les modules dépendants
- `go get -u <modulepath>` Mets à jour un module
- `go mod why` indique le chemin de dépendance entre le package courant et un package (ou un module avec le flag `-m`)
- `go list -m all` affiche la liste des modules à compiler

🎓 Exercice: Compiler le serveur

A vous de jouer, il nous faut maintenant compiler et lancer notre serveur!

✅ Solution: Compiler le serveur

```
# On démarre le serveur de base de données
make dev_db
# On compile le serveur
make all
# On lance notre serveur
./dist/server
```

Copy

Checkpoint 🎯

- La gestion de dépendances est une question importante et complexe
- Une bonne compréhension de son système de gestion de dépendances est nécessaire pour garantir la reproductibilité de nos livrables
- go fournit une suite d'outils intégrées qui simplifie la gestion des dépendances externes
- 🎉 On est maintenant en mesure de compiler notre projet! 🎉

Mettre son code en sécurité

Une autre petite histoire

Votre dépôt est actuellement sur votre ordinateur.

- Que se passe t'il si :
  - Votre disque dur tombe en panne ?
  - On vous vole votre ordinateur ?
  - Vous échappez votre tasse de thé / café sur votre ordinateur ?
  - Une météorite tombe sur votre bureau et fracasse votre ordinateur ?

Testé, pas approuvé.

Comment éviter ça ?

- Répliquer votre dépôt sur une ou plusieurs machines !
- Git est pensé pour gérer ce de problème

Gestion de version décentralisée

- Chaque utilisateur maintient une version du dépôt *local* qu'il peut changer à souhait
- Ils peuvent "pousser" une version sur un dépôt **distant**
- Un dépôt *local* peut avoir plusieurs dépôts **distants**.

Centralise vs Decentralise

Source Geek for Geeks

Créer un dépôt distant

- Rendez vous sur GitHub
  - Créez un nouveau dépôt distant en cliquant sur "New" en haut à gauche
  - Appelez le `vehicle-server`
  - Une fois créé, mémorisez l'URL (https://github.com/...) de votre dépôt :-)
  - Inscrivez l'URL de votre depot ici.

Consulter l'historique de commits

Dans votre workspace

```
# Liste tous les commits présent sur la branche main.
git log
```

Copy

Associer un dépôt distant (1/2)

Git permet de manipuler des "remotes"

- Image "distante" (sur un autre ordinateur) de votre dépôt local.
- Permet de publier et de rapatrier des branches.
- Le serveur maintient sa propre arborescence de commits, tout comme votre dépôt local.
- Un dépôt peut posséder N remotes.

Associer un dépôt distant (2/2)

```
# Liste les remotes associés a votre dépôt
git remote -v

# Ajoute votre dépôt comme remote appelé `origin`
git remote add origin https://<URL de votre dépôt>

# Vérifiez que votre nouveau remote `origin` est bien listé a la bonne adresse
git remote -v
```

Copy

Publier une branche dans sur dépôt distant

Maintenant qu'on a un dépôt, il faut publier notre code dessus !

```
# git push <remote> <votre_branche_courante>
git push origin main
```

Copy

Que s'est il passé ?

- `git` a envoyé la branche `main` sur le remote `origin`
- … qui à accepté le changement et mis à jour sa propre branche main.
- `git` a créé localement une branche distante `origin/main` qui suis l'état de `main` sur le remote.
- Vous pouvez constater que la page github de votre dépôt affiche le code source

Refaisons un commit !

```
git commit --allow-empty -m "Yet another commit"
git push origin main
```

Copy

remotes/origin/main
main
Push
main
Local
Origin

Branche distante

Dans votre dépôt local, une branche "distante" est automatiquement maintenue par git

C’est une image du dernier état connu de la branche sur le remote.

Pour mettre a jour les branches distantes depuis le remote il faut utiliser :

```
git fetch <nom_du_remote>
```

```
# Lister toutes les branches y compris les branches distances
git branch -a


# Notez que est listé remotes/origin/main


# Mets a jour les branches distantes du remote origin
git fetch origin


# Rien ne se passe, votre dépôt est tout neuf, changeons ça!
```

Copy

Créez un commit depuis GitHub directement

- Cliquez sur le bouton éditer en haut à droite du "README"
- Changez le contenu de votre README
- Dans la section "Commit changes"
  - Ajoutez un titre de commit et une description
  - Cochez "Commit directly to the main branch"
  - Validez

GitHub crée directement un commit sur la branche main sur le dépôt distant

Rapatrier les changements distants

```
# Mets à jour les branches distantes du dépôt origin
git fetch origin

# La branche distante main a avancé sur le remote origin
# => La branche remotes/origin/main est donc mise a jour

# Ouvrez votre README
code ./README.md

# Mystère, le fichier README ne contient pas vos derniers changements?
git log

# Votre nouveau commit n'est pas présent, AHA !
```

Copy

remotes/origin/main
main
Fetch
main
Local
Origin

Branche Distante VS Branche Locale

Le changement à été rapatrié, cependant il n’est pas encore présent sur votre branche main locale

```
# Merge la branch distante dans la branche locale.
git merge origin/main
```

Copy

Vu que votre branche main n’a pas divergé (== partage le même historique) de la branche distante, `git merge` effectue automatiquement un "fast forward".

```
Updating 1919673..b712a8e
Fast-forward
 README.md | 1 +

 1 file changed, 1 insertion(+)
```

Cela signifie qu’il fait "avancer" la branche `main` sur le même commit que la branche `origin/main`

remotes/origin/main
main
Fetch
main
Local
Origin

```
# Liste l'historique de commit
git log

# Votre nouveau commit est présent sur la branche main !
# Juste au dessus de votre commit initial !
```

Copy

Et vous devriez voir votre changement dans le ficher README.md

Git(Hub|Lab|tea|…)

Un dépôt distant peut être hébergé par n'importe quel serveur sans besoin autre qu'un accès SSH ou HTTPS.

Une multitudes de services facilitent et enrichissent encore git: (GitHub, Gitlab, Gitea, Bitbucket…)

⇒ Dans le cadre du cours, nous allons utiliser **GitHub**.

git + Git(Hub|Lab|tea|…) = superpowers !

- GUI de navigation dans le code
- Plateforme de gestion et suivi d’issues
- Plateforme de revue de code
- Intégration aux moteurs de CI/CD
- And so much more…

Intégration Continue (CI)

> *Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.*

— Martin Fowler

Pourquoi la CI ?

But : Détecter les fautes au plus tôt pour en limiter le coût

Source : http://cartoontester.blogspot.be/2010/01/big-bugs.html

Qu’est ce que l’Intégration Continue ?

Objectif : que l’intégration de code soit un *non-évènement*

- Construire et intégrer le code **en continu**
- Le code est intégré **souvent** (au moins quotidiennement)
- Chaque intégration est validée par une exécution **automatisée**

Et concrètement ? 1/2

- Un•e développeu•se•r ajoute du code/branche/PR :
  - une requête HTTP est envoyée au système de "CI"
- Le système de CI compile et teste le code
- On ferme la boucle : Le résultat est renvoyé au développeu•se•r•s

Et concrètement ? 2/2

Quelques moteurs de CI connus

- A héberger soit-même : Jenkins, GitLab, Drone CI, CDS…
- Hébergés en ligne : Travis CI, Semaphore CI, Circle CI, Codefresh, GitHub Actions

GitHub Actions

GitHub Actions est un moteur de CI/CD intégré à GitHub

- ✅ : Très facile à mettre en place, gratuit et intégré complètement
- ❌ : Utilisable uniquement avec GitHub, et DANS la plateforme GitHub

Concepts de GitHub Actions

Concepts de GitHub Actions - Step 1/3

Concepts de GitHub Actions - Step 2/3

Une **Step** (étape) est une tâche individuelle à faire effectuer par le CI :

- Par défaut c'est une commande à exécuter - mot clef `run`
- Ou une "action" (quel est le nom du produit déjà ?) - mot clef `uses`
  - Réutilisables et partageables

```yaml
steps: # Liste de steps
  # Exemple de step 1 (commande)
  - name: Say Hello
    run: echo "Hello ENSG"
  # Exemple de step 2 (une action)
  - name: 'Login to DockerHub'
    uses: docker/login-action@v1 # https://github.com/marketplace/actions/docker-login
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
```

Copy

Concepts de GitHub Actions - Step 3/3

Une **Step** peut avoir des outputs

Copy

```yaml
steps:
  - name: "Install Go"
    uses: actions/setup-go@v5
    id: setup_go
    with:
      go-version: '1.22'

  - name: "Echo installed version"
    run |
      echo "${{steps.setup_go.outputs.go-version}}"
```

Concepts de GitHub Actions - Job 1/2

Concepts de GitHub Actions - Job 2/2

Un **Job** est un groupe logique de steps :

- Enchaînement *séquentiel* de steps
- Regroupement logique : "qui a un sens"
  - Exemple : "compiler puis tester le résultat de la compilation"

```yaml
jobs: # Map de jobs
  build: # 1er job, identifié comme 'build'
    name: 'Build Slides'
    runs-on: ubuntu-22.04 # cf. prochaine slide "Concepts de GitHub Actions - Runner"
    steps: # Collection de steps du job
      - name: 'Build the JAR'
        run: mvn package
      - name: 'Run Tests on the JAR file'
        run: mvn verify
  deploy: # 2nd job, identifié comme 'deploy'
    # ...
```

Copy

Concepts de GitHub Actions - Runner

Un **Runner** est un serveur distant sur lequel s'exécute un job.

- Mot clef `runs-on` dans la définition d'un job
- Défaut : machine virtuelle Ubuntu dans le cloud utilisé par GitHub
- D'autres types sont disponibles (macOS, Windows, etc.)
- Possibilité de fournir son propre serveur

Concepts de GitHub Actions - Workflow 1/2

Concepts de GitHub Actions - Workflow 2/2

Un **Workflow** est une procédure automatisée composée de plusieurs jobs, décrite par un fichier YAML.

- On parle de "Workflow/Pipeline as Code"
- Chemin : `.github/workflows/<nom du workflow>.yml`
- On peut avoir *plusieurs* fichiers donc *plusieurs* workflows

```
.github/workflows
├── ci-cd.yaml
├── bump-dependency.yml
└── nightly-tests.yaml
```

Copy

Concepts de GitHub Actions - Évènement 1/2

Concepts de GitHub Actions - Évènement 2/2

Un **évènement** du projet GitHub (push, merge, nouvelle issue, etc.) déclenche l'exécution du workflow

- Plein de type d'évènements : push, issue, alarme régulière, favori, fork, etc.
  - Exemple : "Nouveau commit poussé", "chaque dimanche à 07:00", "une issue a été ouverte" …
- Un workflow spécifie le(s) évènement(s) qui déclenche(nt) son exécution
  - Exemple : "exécuter le workflow lorsque un nouveau commit est poussé ou chaque jour à 05:00 par défaut"

Concepts de GitHub Actions : Exemple Complet

Workflow File :

```yaml
name: Node.js CI
on: # Évènements déclencheurs
  - push:
      branch: main # Lorsqu'un nouveau commit est poussé sur la branche "main"
  - schedule:
    - cron: "*/15 * * * *" # Toutes les 15 minutes
jobs:
  test-linux:
    runs-on: ubuntu-22.04
    steps:
    - uses: actions/checkout@v4
    - run: npm install
    - run: npm test
  test-mac:
    runs-on: macos-12
    steps:
    - uses: actions/checkout@v4
    - run: npm install
    - run: npm test
```

Copy

Essayons GitHub Actions

- **But** : nous allons créer notre premier workflow dans GitHub Actions
- N'hésitez pas à utiliser la documentation de GitHub Actions:
  - Accueil
  - Quickstart
  - Référence
- Retournez dans le dépôt créé précédemment dans votre environnement GitPod

Exemple simple avec GitHub Actions

- Dans le projet "vehicle-server", sur la branch `main`,
  - Créez le fichier `.github/workflows/bonjour.yml` avec le contenu suivant :

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Bonjour 👋"
```

Copy

- Commitez puis poussez
- Revenez sur la page GitHub de votre projet et naviguez dans l'onglet "Actions" :
  - Voyez-vous un workflow ? Et un Job ? Et le message affiché par la commande `echo` ?

Exemple simple avec GitHub Actions : Récapépète

Exemple GitHub Actions : Checkout

- Supposons que l'on souhaite utiliser le code du dépôt…
  - Essayez: modifiez le fichier `bonjour.yml` pour afficher le contenu de `README.md` :

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: ls -l # Liste les fichier du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

- Est-ce que l'étape se passe bien ? (SPOILER: non ✖)

🎓 Exercice GitHub Actions : Checkout

- **But** : On souhaite récupérer ("checkout") le code du dépôt dans le job
- 👷‍♀️ C’est à vous d’essayer de *réparer* 🛠️ le job :
  - L’étape doit être conservée et doit fonctionner
  - Utilisez l’action "checkout" (Documentation) du marketplace GitHub Action
  - Vous pouvez vous inspirer du Quickstart de GitHub Actions

✅ Solution GitHub Actions : Checkout

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: ls -l # Liste les fichier du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

Copy

Exemple : Environnement d'exécution

- Notre workflow doit s'assurer que "la vache" 🐮 doit nous lire 💬 le contenu du fichier `README.md`
  - WAT 🤪 ?
  - Essayez la commande `cat README.md | cowsay` dans GitPod
    - Modifiez l'étape du workflow pour faire la même chose dans GitHub Actions
    - SPOILER: ❌ (la commande `cowsay` n'est pas disponible dans le runner GitHub Actions)

Problème : Environnement d'exécution

- **Problème** : On souhaite utiliser les mêmes outils dans notre workflow ainsi que dans nos environnement de développement
- Plusieurs solutions existent pour personnaliser l'outillage, chacune avec ses avantages / inconvénients :
  - Personnaliser l'environnement dans votre workflow: (⚠ sensible aux mises à jour, ✅ facile à mettre en place)
  - Spécifier un environnement préfabriqué pour le workflow (⚠ complexe, ✅ portable)
  - Utiliser les fonctionnalités de votre outil de CI (⚠ spécifique au moteur de CI, ✅ efficacité)

🎓 Exercice : Personnalisation dans le workflow

- **But** : exécuter la commande `cat README.md | cowsay` dans le workflow comme dans GitPod
- 👩‍🏗️ C'est à vous de mettre à jour le workflow pour personnaliser l'environnement :
  - 💡 Cherchez comment installer `cowsay` dans le runner GitHub (`runs-on`, paquet `cowsay` dans Ubuntu 22.04)

✅ Solution : Personnalisation dans le workflow

Copy

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: |
          sudo apt-get update
          sudo apt-get install -y cowsay
      - run: cat README.md | cowsay
```

🎓 Exercice : Environnement préfabriqué

- **But** : exécuter la commande `cat README.md | cowsay` dans le workflow comme dans GitPod
  - En utilisant le même environnement que GitPod (même version de cowsay, java, etc.)
- 👷‍♀️ C'est à vous de mettre à jour le workflow pour exécuter les étapes dans la même image Docker que GitPod :
  - 💡 Image utilisée dans GitPod
  - 💡 Utilisation d'un container comme runner GitHub Actions
  - 💡 Contraintes d'exécution de container dans GitHub Actions (`--user=root`)

✅ Solution : Environnement préfabriqué

Copy

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    container:
      image: ghcr.io/cicd-lectures/gitpod:latest
      options: --user=root
    steps:
      - uses: actions/checkout@v4 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: cat README.md | cowsay
```

Checkpoint 🎯

- Quel est l'impact en terme de temps d'exécution du changement précédent ?
- **Problème :** Le temps entre une modification et le retour est crucial

du moteur de CI

- **But** : s'assurer que GitHub actions install et utilise `cowsay` le plus efficacement possible
- C'est à vous de mettre à jour le workflow pour:
  - Lire le contenu du fichier `README.md` dans un "output" (une variable temporaire de GitHub Actions)
  - Passer le contenu (via l'output) à une version de cowsay gérée par GitHub Actions
- 💡 Utilisez les GitHub Actions et documentations suivantes :
  - GitHub Action pour cowsay
  - GitHub Action pour lire un fichier dans une variable `output`

✅ Solution : Optimiser avec les fonctionnalités du moteur de CI

Copy

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - uses: juliangruber/read-file-action@v1
        id: readfile
        with:
          path: ./README.md
      - uses: Code-Hex/neo-cowsay-action@v1
        with:
          message: "${{ steps.readfile.outputs.content }}"
```

🎓 Exercice : Intégration Continue du projet "vehicle-server"

👷🏽‍♀️ C'est à vous de modifier le projet "vehicle-server" pour faire l'intégration continue, afin qu'à chaque commit poussé sur votre dépôt, un workflow GitHub Actions va :

- Récupérer le code de l'application depuis GitHub
- Installer go dans la version spécifiée par le fichier `go.mod`
  - 💡 https://github.com/actions/setup-go
- L'application est compilée et le binaire est généré dans dist
- Pensez à supprimer/renommer le workflow `bonjour.yaml`

✅ Solution : Intégration Continue du projet "vehicle-server"

Copy

```yaml
name: Vehicle Server CI
on:
  - push
jobs:
  ci:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4
      - name: Setup Go
      - uses: actions/setup-go@v5
        with:
          go-version-file: './go.mod'
      - name: Check Go Version
        run: go version
      - name: Build application
        run: make build
      - name: List dist output
        run: ls dist/
```

🎯 Checkpoint

- Pour chaque commit poussé dans la branche `main` du Vehicle Server,
- GitHub action vérifie que l'application est compilable et fabriquée,
- Avec un feedback (notification GitHub).

⇒ On peut modifier notre code avec plus de confiance !

Git à plusieurs

Limites de travailler seul

- Capacité finie de travail
- Victime de propres biais
- On ne sait pas tout

Travailler en équipe ? Une si bonne idée ?

- … Mais il faut communiquer ?
- … Mais tout le monde n’a pas les mêmes compétences ?
- … Mais tout le monde y code pas pareil ?

Collaborer c’est pas évident, mais il existe des outils et des méthodes pour vous aider.

Cela reste des outils, ça ne résous pas tout non plus.

Git multijoueur

- Git permet de collaborer assez aisément
- Chaque développeur crée et publie des commits…
- … et rapatrie ceux de de ses camarades !
- C’est un outil très flexible… chacun peut faire ce qu’il lui semble bon !

BLOING!
BLOING!

Un Example de Git Flow

(Attachez vous aux idées générales… les détails varient d’un projet à l’autre!)

Gestion des branches

- Les "versions" du logiciel sont maintenues sur des branches principales (main, staging)
- Ces branches reflètent l’état du logiciel
  - **main**: version actuelle en production
  - **staging**: prochaine version

staging
main
Origin

Gestion des branches

- Chaque groupe de travail (développeur, binôme…)
  - Crée une branche de travail à partir de la branche staging
  - Une branche de travail correspond à **une chose à la fois**
  - Pousse des commits dessus qui implémentent le changement

Origin

Quand le travail est fini, la branche de travail est "mergée" dans staging

main
staging

Gestion des remotes

Où vivent ces branches ?

Plusieurs modèles possibles

- Un remote pour les gouverner tous !
- Chacun son propre remote (et les commits seront bien gardés)
- … whatever floats your boat!

Un remote pour les gouverner tous

Tous les développeurs envoient leur commits et branches sur le même remote

- Simple a gérer …
- … mais nécessite que tous les contributeurs aient accès au dépôt
  - Adapté a l'entreprise, peu adapté au monde de l'open source

https://github.com/torvalds/linux.git
upstream

Chacun son propre remote

- La motivation: le contrôle d’accès
  - Tout le monde peut lire le dépôt principal. Personne ne peut écrire dessus.
  - Tout le monde peut dupliquer le dépôt public et écrire sur sa copie.
  - Toute modification du dépôt principal passe par une procédure de revue.
  - Si la revue est validée, alors la branche est "mergée" dans la branche cible
- C’est le modèle poussé par GitHub !

https://github.com/torvalds/linux.git
upstream
Pull Request
https://github.com/jlevesy/linux.git
origin
Torvalds
jlevesy

Forks ! Forks everywhere !

Dans la terminologie GitHub:

- Un fork est un remote copié d'un dépôt principal
  - C'est la où les contributeurs poussent leur branche de travail.
- Les branches de version (main, staging…) vivent sur le dépôt principal
- La procédure de ramener un changement d'un fork à un dépôt principal s'appelle la Pull Request (PR)

- Nous allons vous faire forker vos dépôts respectifs
- Trouvez vous un binôme dans le groupe.
- Rendez vous sur cette page pour inscrire votre binôme.
- Depuis la page du dépôt de votre binôme, cliquez en haut à droite sur le bouton **Fork**.

A vous de jouer: Corrigez la fonctionnalité "suppression d’un véhicule" dans projet de votre binôme

🎓 Exercice : Contribuez au projet de votre binôme (1/4)

Première étape: on clone le fork dans son environnement de développement

```
cd /workspace/

# Clonez votre fork
git clone <url_de_votre_fork>

# Créez votre feature branch
git switch --create fix-delete
# Équivalent de git checkout -b <...>
```

Copy

Minute Go: L'interface http.Handler

```go
type Handler interface {
        ServeHTTP(rw ResponseWriter, r *Request)
}
```

Copy

- Interface de la librairie standard qui normalise tous les types pouvant "gérer une requête HTTP"
- Elle accepte deux arguments
  - `rw`: Le "response writer", qui permet d'écrire le code de statut, les headers et le corps de la réponse
  - `r`: La requête à traiter
- Nos implémentations de `http.Handler` se trouvent dans le package `vehicle`.

Pouvez vous identifier le code problématique?

binôme (2/4)

- Extraire l'identifiant (la valeur `id`) du path en utilisant la méthode PathValue.
- Parser la valeur obtenue en `int64` en utilisant strconv.ParseInt
- Appeler la methode `Delete` du `VehicleStore` en passant l'identifiant et le "contexte" récupéré en appelant `r.Context()`
- Enfin il faut faire une réponse:
  - Si la suppression est réussie, répondre un status code 204 en appelant `rw.WriteHeader(http.StatusNoContent)`,
  - Si le véhicle indiqué n'existe pas, répondre 404.
- Il convient de gérer les erreurs correctement, vous pouvez vous inspirer des beautres handlers défini

🎓 Exercice : Contribuez au projet de votre binôme (3/4)

Une fois que vous êtes satisfaits de votre changement il vous faut maintenant créer un commit et pousser votre nouvelle branche sur votre fork.

binôme (4/4)

Dernière étape: ouvrir une pull request!

- Rendez vous sur la page de votre projet
- Sélectionnez votre branche dans le menu déroulant "branches" en haut a gauche.
- Cliquez ensuite sur le bouton ouvrir une pull request
- Remplissez le contenu de votre PR (titre, description, labels) et validez.

La procédure de Pull Request

Objectif : Valider les changements d'un contributeur

- Technique : est-ce que ça marche ? Est-ce maintenable ?
- Fonctionnel : est-ce que le code fait ce que l'on veux ?
- Humain : Propager la connaissance par la revue de code.
- Méthode : Tracer les changements.

Revue de code ?

- Validation par un ou plusieurs pairs (technique et non technique) des changements
- Relecture depuis github (ou depuis le poste du développeur)
- Chaque relecteur émet des commentaires // suggestions de changement
- Quand un relecteur est satisfait d'un changement, il l'approuve

- La revue de code est un **exercice difficile** et **potentiellement frustrant** pour les deux parties.
  - Comme sur Twitter, on est bien à l'abri derrière son écran
- En tant que contributeur, **soyez respectueux** de vos relecteurs : votre changement peut être refusé et c'est quelque chose de normal.
- En tant que relecteur, **soyez respectueux** du travail effectué, même si celui ci comporte des erreurs ou ne correspond pas à vos attentes.

💡 Astuce: Proposez des solutions plutôt que simplement pointer les problèmes.

🎓 Exercice : Relisez votre PR reçue !

- Vous devriez avoir reçu une PR de votre binôme
- Relisez le changement de la PR
- Effectuez quelques commentaires (bonus: utilisez la suggestion de changements), si c’est nécessaire
- Si elle vous convient, approuvez la!
- En revanche ne la "mergez" pas, car il manque quelque chose…

Validation automatisée

Objectif: Valider que le changement n’introduit pas de régressions dans le projet

- A chaque fois qu’un nouveau commit est créé dans une PR, une succession de validations ("checks") sont déclenchés par GitHub
- Effectue des vérifications automatisées sur un commit de merge entre votre branche cible et la branche de PR

Quelques exemples

- Analyse syntaxique du code (lint), pour détecter les erreurs potentielles ou les violations du guide de style
- Compilation du projet
- Exécution des tests automatisés du projet
- Déploiement du projet dans un environnement de test…

Ces "checks" peuvent êtres exécutés par votre moteur de CI ou des outils externes.

🎓 Exercice : Déclencher un Workflow de CI sur une PR

- Votre PR n'a pas déclenché le workflow de CI de votre binôme 🤔
- Il faut changer la configuration de votre workflow pour qu'il se déclenche aussi sur une PR
- Vous pouvez changer la configuration du workflow directement dans votre PR
- La documentation se trouve par ici

⚠ Ne mergez toujours pas votre PR ⚠

✅ Exercice : Déclencher un Workflow de CI sur une PR

Copy

```yaml
name: Vehicle Server CI
on:
  - push
  - pull_request
jobs:
  ci:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4
      - name: Setup Go
      - uses: actions/setup-go@v5
        with:
          go-version-file: './go.mod'
      - name: Check Go Version
        run: go version
      - name: Build application
        run: make build
      - name: List dist output
        run: ls dist/
```

Règle d'or: Si le CI est rouge, on ne merge pas la pull request !

🎯 Checkpoint

Nous avons vu:

- Un exemple de modèle de gestion de branches git
- Plusieurs modèles de gestions de remotes
- Ce qu’est un "fork" sur GitHub
- Le processus pour effectuer une contribution en utilisant un projet forké!

➡ Nous allons maintenant améliorer notre CI en ajoutant l’exécution des tests!

Tests Automatisés

Qu’est ce qu’un test ?

C’est du code qui vérifie que votre code fait ce qu’il est supposé faire.

Pourquoi faire des tests ?

- Prouve que le logiciel se comporte comme attendu à tout moment
- Détecte les impacts non anticipés des changements introduits

Qu’est ce que l’on teste ?

- Une fonction
- Une combinaison de classes
- Un serveur applicatif et une base de données

On parle de **SUT**, System Under Test.

Différents systèmes, Différentes Techniques de Tests

- Test unitaire
- Test d'intégration
- Test de bout en bout
- Smoke tests
- Test de performance

(La terminologie varie d'un développeur / langage / entreprise / écosystème à l'autre)

Test unitaire

- Test validant le bon comportement une unité de code
- Prouve que l’unité de code interagit correctement avec les autres unités.
- Les autres composants dont l’unité de code dépends sont "bouchonnés", cela pour garantir leur simplicité et leur facilité.
  - Par Exemple: la couche d’accès a la base de données est réimplémentée en mémoire.

Tests & Go

- Go embarque un framework de tests intégré a la CLI `go`, la commande `go test`
- Les tests sont écrit dans un fichier `_test.go` apposé au fichier testé
- `go test` découvre touts les fichiers de tests d'un module et joue tous les tests implémentés dans ces fichiers.
  - Un test est une fonction qui commence par `Test` et qui accepte en paramètre un pointeur sur `testing.T`

```go
func TestToStringSlice(t *testing.T) {
	// Given.
	input := []int{1, 2, 3, 4}

	// When.
	got := ToStringSlice(input)

	// Then.
	want := []string{"1", "2", "3", "4"}

	// On vérifie que la longueur récupérée est conforme à celle attendue.
	if len(want) != len(got) {
		t.Fail("Mismatched length")
		return
	}

	// Pour chacune des entrées récupérées
	// On vérifie que l'entrée correspond à la valeur attendue.
	for i, gotValue := range got {
		if want[i] != gotValue {
			t.Fail()
		}
	}
}
```

Copy

Tests Paramétriques

- On veut tester la même méthode avec plusieurs valeurs d'entrées
- Cela crée pas mal de duplication
- **Solution**: On utilise des tests paramétriques qui factorisent le code du test et variabilise les valeurs d'entrées et les valeurs attendues

```go
func TestToStringSlice(t *testing.T) {
	// On crée un tableau de cas de tests
	testCases := []struct {
		name  string
		input []int
		want  []string
	}{
		{
			name:   "normal case",
			input:  []int{1, 2, 3, 4},
			output: []string{"1", "2", "3", "4"},
		},
		{
			name:   "empty case",
			input:  nil,
			output: nil,
		},
	}

	// Pour chacun de ces cas de tests.
	for _, testCase := range testCases {
		// On execute un sous test qui utilises les attributs
		// du cas de test.
		t.Run(testCase.name, func(t *testing.T) {
			got := ToStringSlice(testCase.input)

			if len(testCase.want) != len(got) {
				t.Fail("Mismatched length")
				return
```

Exécutons les Tests du Package vehicle

On exécute les tests d'un package particulier avec la commande suivante.

```
go test -v ./vehicle
```

Copy

Quel est le résultat?

🤔 Il semblerait que quelque chose ne se passe pas comme prévu…

🎓 Exercice: Corrigez Le Bug 🐛

- À l'aide de la sortie du test, essayez de déterminer d'où peut venir le problème.
- Quelques questions pour vous aider:
  - Quel méthode est testée? Dans quel fichier se trouve t'elle? Que doit faire cette méthode?
  - Quel comportement est testé par le cas de test?
  - Quel code HTTP est attendu? Retourné? Que signifient t'ils?
- Une fois le problème clairement identifié, proposez un changement qui valide ce cas de test.

⚠ On ne change / supprime pas le code du test ⚠

✅ Solution: Corrigez Le Bug 🐛

```go
// vehicle/create.go:126

if len(f.ShortCode) > 4 {
  validationIssues = append(validationIssues, "short code too long")
}
```

Copy

🎓 Exercice: Ajoutez La Target `unit_test` au Makefile

- Pour faciliter l'exécution de toute la suite de tests, ajoutez la cible `unit_test` à votre Makefile pour que les tests de tous les packages soit joués.
- Avec le mode verbeux activé par défaut
  - Indice: `go help test` et `go help packages`
- **Bonus**: Activez le calcul de la couverture de tests

✅ Solution: Ajoutez La Target `unit_test` au Makefile

```
.PHONY: unit_test
unit_test:
  go test -v -cover ./...
```

Copy

Test Unitaire : Pro / Cons

- ✅ Super rapides (<1s) et légers a exécuter
- ✅ Pousse à avoir un bon design de code
- ✅ Efficaces pour tester des cas limites
- ❌ Environnement "aseptisé" et "bouchonné", défini par le développeur
- ❌ "Ossifie" le code

Le périmètre testé est-il satisfaisant?

- La suite de tests qui vient de casser teste la logique de validation de la requête reçue.
- Est-ce que cela est suffisant pour prouver que la fonctionnalité "créer un véhicule" fonctionne ?

- Pas exactement, d'autres composants entrent en jeu dans l'environnement réel
  - La couche de communication avec la base de données, le routage HTTP…

Tester des composants indépendamment ne prouve pas que le système fonctionne une fois intégré!

✅ Solution: Tests d'intégration

- Test validant que l'assemblage de composants se comportent comme prévu.
- Teste votre application au travers de tous ses composants
- Par exemple avec vehicle-server:
  - Prouve que GET /vehicles retourne la liste des véhicules les plus proche d'un point donné
  - Prouve que POST /vehicles enregistre un nouveau véhicule en base.

Définition du SUT

Une suite de tests d'intégration doit:

- Démarrer et provisionner un environnement d'exécution (une DB, Elasticsearch, un autre service…)
- Démarrer votre application
- Jouer un scénario de test
- Éteindre et nettoyer son environnement d'exécution pour garantir l'isolation des tests

➡ On se place ici d'un point de vue extérieur à l'application

- ✕ Ce sont des tests plus lents et plus complexes que des tests unitaires.
- ⏳ Tout tester avec des tests d'intégration n'est pas efficace
- ➡ Il faut équilibrer les deux stratégies

Et concrètement avec le notre projet?

- La suite de test d'intégration se situe dans le fichier `app_test.go`
- Il y a un cas par fonctionnalité principale de l'application (create vehicle, delete vehicle, list vehicles)

- La gestion de l’environnement est "cachée":
  - Démarrer un container de base de données (Postgres avec Postgis)
  - Démarrer une instance de l’application
  - Tout éteindre une fois terminé

C’est fait dans le fichier `app/helper_test.go` si vous voulez 👀

Lancez les tests d'intégration

- Vous pouvez lancer les tests d'intégration avec la commande suivante.

```
go test -v -count=1 --tags=integration ./app
```

Copy

🎓 Exercice: Corrigez votre Implémentation de Delete Vehicle

Optionnel mais, si les tests d'intégration échouent à ce point, il serait bon de corriger votre implémentation de la suppression de véhicule.

🎓 Exercice: Activez les tests dans votre CI

Changez le workflow de ci de votre binôme (ou le votre) pour qu'à chaque build:

- Les tests unitaires soient lancés
- Les tests d'intégration soient lancés
  - N'oubliez pas de définir une nouvelle cible integration_test dans le Makefile

✅ Solution: Activez les tests dans votre CI

Copy

```yaml
name: Vehicle Server CI
on:
  - push
  - pull_request
jobs:
  ci:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4
      - name: Setup Go
      - uses: actions/setup-go@v5
        with:
          go-version-file: './go.mod'
      - name: Check Go Version
        run: go version
      - name: Run Unit Tests
        run: make unit_test
      - name: Run Integration Tests
        run: make integration_test
      - name: Build application
        run: make build
      - name: List dist output
        run: ls dist/
```

Checkpoint 🎯

On a vu :

- ✖ Les avantages et limites des différentes stratégies de tests…
- ⚖️ … et la nécessité d'avoir une stratégie équilibrée.

🎉 Vous pouvez enfin merger votre PR!

Docker

Remise à niveau / Rappels

🤔 Quel est le problème ?

	Development VM	QA Server	Single Prod Server	Production Cluster	Public cloud	Developer's Laptop	Customer Servers
Static Website	?	?	?	?	?	?	?
Web Frontend	?	?	?	?	?	?	?
Background Workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Source: https://blog.docker.com/2013/08/paas-present-and-future/

Déjà vu ?

L’IT n’est pas la seule industrie à résoudre des problèmes…

✅ Solution: Le conteneur intermodal

"Separation of Concerns"

Comment ça marche ?

"Virtualisation **Légère**"

Conteneur != VM

"Separation of concerns": 1 "tâche" par conteneur

VMs && Conteneurs

Non exclusifs mutuellement

🎓 Exercice : où est mon conteneur ?

- Retournez dans Gitpod
- Dans un terminal, exécutez les commandes suivantes :

Copy

```
# Affichez la liste de tous les conteneurs en fonctionnement (aucun)
docker container ls

# Exécutez un conteneur
docker container run hello-world # Equivalent de l'ancienne commande 'docker run'


docker container ls
docker container ls --all
# Quelles différences ?
```

🩻 Anatomie

- Un service "Docker Engine" tourne en tâche de fond et publie une API REST
- La commande `docker run` ... a envoyé une requête `POST` au service
- Le service a télécharge une **Image** Docker depuis le registre **DockerHub**,
- Puis a exécuté un **conteneur** basé sur cette image

✅ Solution : Où est mon conteneur ?

Le conteneur est toujours présent dans le "Docker Engine" même en étant arrêté

```
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
109a9cdd3ec8   hello-world   "/hello"   33 seconds ago   Exited (0) 17 seconds ago             festive_faraday
```

Copy

- Un conteneur == une commande "conteneurisée"
  - cf. colonne "**COMMAND**"
- Quand la commande s’arrête : le conteneur s’arrête
  - cf. code de sortie dans la colonne "**STATUS**"

Exercice : Cycle de vie d'un conteneur en tâche de fond

- Lancez un nouveau conteneur en tâche de fond, nommé `webserver-1` et basé sur l'image `nginx`
  - 💡 `docker container run --help` ou Documentation en ligne
- Affichez les "logs" du conteneur (==traces d'exécution écrites sur le stdout + stderr de la commande conteneurisée)
  - 💡 `docker container logs --help` ou Documentation en ligne
- Comparez les versions de Linux de Gitpod et du conteneur

? Regardez le contenu du fichier `/etc/os-release`

✅ Solution : Cycle de vie d'un conteneur en tâche de fond

```
docker container run --detach --name=webserver-1 nginx
# <ID du conteneur>

docker container ls

docker container logs webserver-1

cat /etc/os-release
# ... Ubuntu ...
docker container exec webserver-1 cat /etc/os-release
# ... Debian ...
```

Copy

🤔 Comment accéder au serveur web en tâche de fond ?

```
$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS     NAMES
ee5b70fa72c3   nginx     "/docker-entrypoint.…"   3 seconds ago   Up 2 seconds   80/tcp    webserver-1
```

- ✅ Super, le port 80 (TCP) est annoncé (on parle d'"exposé")…
- ❌ … mais c'est sur une adresse IP privée

```
docker container inspect \
  --format='{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
  webserver-1
```

🎓 Exercice : Accéder au serveur web via un port publié

- **But :** Créez un nouveau conteneur `webserver-public` accessible publiquement
- Utilisez le port `8080` publique
- 💡 Flag `--publish` pour `docker container run`
- 💡 GitPod va vous proposer un popup : choisissez "Open Browser"

✅ Solution : Accéder au serveur web via un port publié

```
docker container run --detach --name=webserver-public --publish 8080:80 nginx
# ... container ID ...

docker container ls
# Le port 8080 de 0.0.0.0 est mappé sur le 80 du conteneur

curl http://localhost:8080
# ...
```

Copy

🤔 D’où vient "hello-world" ?

- Docker Hub (https://hub.docker.com) : C’est le registre d’images "par défaut"
  - Exemple : Image officielle de "nginx"
- 🎓 Cherchez l’image `hello-world` pour en voir la page de documentation
  - 💡 pas besoin de créer de compte pour ça
- Il existe d’autre "registres" en fonction des besoins (GitHub GHCR, Google GCR, etc.)

🤔 Que contient "hello-world" ?

- C’est une "image" de conteneur, c’est à dire un modèle (template) représentant une application auto-suffisante.
  - On peut voir ça comme un "paquetage" autonome
- C’est un système de fichier complet:
  - Il y a au moins une racine `/`
  - Ne contient que ce qui est censé être nécessaire (dépendances, librairies, binaires, etc.)

🤔 Pourquoi des images ?

- Un **conteneur** est toujours exécuté depuis une **image**.
- Une **image de conteneur** (ou "Image Docker") est un modèle ("template") d'application auto-suffisant.

⇒ Permet de fournir un livrable portable (ou presque).

🤔 Application Auto-Suffisante ?

C’est quoi le principe ?

🤔 Pourquoi fabriquer sa propre image ?

Essayez ces commandes dans Gitpod :

```
cat /etc/os-release
# ...
git --version
# ...

# Même version de Linux que dans GitPod
docker container run --rm ubuntu:20.04 git --version
# docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable t

# En interactif ?
docker container run --rm --tty --interactive ubuntu:20.04 git --version
```

Copy

⇒ Problème : git n'est même pas présent !

🎓 Fabriquer sa première image

- **But :** fabriquer une image Docker qui contient `git`
- Dans votre workspace Gitpod, créez un nouveau dossier `/workspace/docker-git/`
- Dans ce dossier, créer un fichier `Dockerfile` avec le contenu ci-dessous :

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
```

Copy

- Fabriquez votre image avec la commande `docker image build --tag=docker-git <chemin/vers/docker-git/`
- Testez l'image fraîchement fabriquée

? 💡 `docker image ls`

✅ Fabriquer sa première image

```
mkdir -p /workspace/docker-git/ && cd /workspace/docker-git/

cat <<EOF >Dockerfile
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
EOF

docker image build --tag=docker-git ./

docker image ls | grep docker-git

# Doit fonctionner
docker container run --rm docker-git:latest git --version
```

Copy

Conventions de nommage des images

```
[REGISTRY/][NAMESPACE/]NAME[:TAG|@DIGEST]
```

Copy

- Pas de Registre ? Défaut: `registry.docker.com`
- Pas de Namespace ? Défaut: `library`
- Pas de tag ? Valeur par défaut: `latest`
  - ⚠ Friends don't let friends use `latest`
- Digest: signature unique basée sur le contenu

Conventions de nommage : Exemples

- `ubuntu:20.04` ⇒ `registry.docker.com/library/ubuntu:20.04`
- `dduportal/docker-asciidoctor` ⇒ `registry.docker.com/dduportal/docker-asciidoctor:latest`
- `ghcr.io/dduportal/docker-asciidoctor:1.3.2@sha256:xxxx`

🎓 Utilisons les tags

- Il est temps de "taguer" votre première image !

```
docker image tag docker-git:latest docker-git:1.0.0
```

Copy

- Testez le fonctionnement avec le nouveau tag
- Comparez les 2 images dans la sortie de `docker image ls`

✅ Utilisons les tags

```
docker image tag docker-git:latest docker-git:1.0.0

# 2 lignes
docker image ls | grep docker-git
# 1 ligne
docker image ls | grep docker-git | grep latest
# 1 ligne
docker image ls | grep docker-git | grep '1.0.0'

# Doit fonctionner
docker container run --rm docker-git:1.0.0 git --version
```

Copy

🎓 Mettre à jour votre image (1.1.0)

- Mettez à jour votre image en version `1.1.0` avec les changements suivants :
  - Ajoutez un `LABEL` dont la clef est `description` (et la valeur de votre choix)
  - Configurez `git` pour utiliser une branche `main` par défaut au lieu de `master` (commande `git config --global init.defaultBranch main`)
- Indices :
  - 💡 Commande `docker image inspect <image name>`
  - 💡 Commande `git config --get init.defaultBranch` (dans le conteneur)
  - 💡 Ajoutez des lignes **à la fin** du `Dockerfile`
  - 💡 Documentation de référence des `Dockerfile`

Mettre à jour votre image (1.1.0)

```
cat ./Dockerfile
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
LABEL description="Une image contenant git préconfiguré"
RUN git config --global init.defaultBranch main

docker image build -t docker-git:1.1.0 ./docker-git/
# Sending build context to Docker daemon  2.048kB
# Step 1/4 : FROM ubuntu:20.04
#  ---> e40cf56b4be3
# Step 2/4 : RUN apt-get update && apt-get install --yes --no-install-recommends git
#  ---> Using cache
#  ---> 926b8d87f128
# Step 3/4 : LABEL description="Une image contenant git préconfiguré"
#  ---> Running in 0695fc62ecc8
# Removing intermediate container 0695fc62ecc8
#  ---> 68c7d4fb8c88
# Step 4/4 : RUN git config --global init.defaultBranch main
#  ---> Running in 7fb54ecf4070
# Removing intermediate container 7fb54ecf4070
#  ---> 2858ff394edb
Successfully built 2858ff394edb
Successfully tagged docker-git:1.1.0

docker container run --rm docker-git:1.0.0 git config --get init.defaultBranch
docker container run --rm docker-git:1.1.0 git config --get init.defaultBranch
```

Copy

?

17.29

🎓 Construire une Image du Vehicle Server

- A partir de l'image de base go Go construisez une image du vehicle-server
- Il vous faut copier les sources avec l'instruction COPY
- Compiler le serveur
- Faire en sorte que le point d'entrée de l'image soit le serveur (en utilisant ENTRYPOINT)
- L'image doit être utilisable avec la commande suivante:

```
docker run --tty --interactive --rm --publish 8080:8080 image:tag -listen-address=:8080 -database-url=${POSTGRES_URL}
```

Copy

✅ Construire une Image du Vehicle Server

```
FROM golang:1.22
COPY ./ /app
WORKDIR /app
RUN go build ./cmd/server
ENTRYPOINT ["/app/server"]
```

Copy

Qu’avons nous construit?

- On part d’une image de base avec le compilateur go
- On copie l’intégralité de nos sources dedans
- On compile le binaire dans l’image

Est-ce efficace?

Regardons maintenant la taille de l’image. Est-ce satisfaisant?

- Quels outils avons nous à notre disposition pour optimiser ça?
  - Changer l'image de base pour embarquer uniquement ce qui est nécessaire dans l'image
  - Tirer partie de notre cycle de vie existant pour ne pas compiler dans la phase de création build de l'image.

Prérequis: Variables Makefile

`make` permets de définir des variables

```
URL=?https://ensg.eu

.PHONY: show_url
show_url:
  echo $(URL)
```

Copy

Les variables peuvent êtres surchargées au moment de l'appel

```
make show_url URL=https://google.com
```

Copy

🎓 Construire une Image du Vehicle Server II

- Partir de l'image de base `gcr.io/distroless/static-debian12`
- Copiez le binaire compilé dans `dst` dans l'image et en faire l'entrypoint
  - Attention à l'utilisation d'ENTRYPOINT avec distroless!
- Changer le Makefile pour:
  - Ajouter une cible `package` qui crée l'image Docker
  - Intrégrer `package` à la cible all.

✅ Construire une Image du Vehicle Server II

Dockerfile

```
FROM gcr.io/distroless/static-debian12
COPY dist/server /app/server
ENTRYPOINT ["/app/server"]
```

Copy

Makefile

```
IMAGE?=<VOTRE_USERNAME_DOCKERHUB>/vehicle-server
TAG?=dev

.PHONY: all
all: clean unit_test integration_test build package

.PHONY: package
package:
  docker build -t $(IMAGE):$(TAG) .
```

Copy

🎓 Ajouter Package au CI

- Il faut tester aussi lors du CI que l'étape package fonctionne à chaque instant
- Ajoutez package au CI

Ajouter l'étape Package au CI

Copy

```yaml
name: Vehicle Server CI
on:
  - push
  - pull_request
jobs:
  ci:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4
      - name: Setup Go
      - uses: actions/setup-go@v5
        with:
          go-version-file: './go.mod'
      - name: Check Go Version
        run: go version
      - name: Run Unit Tests
        run: make unit_test
      - name: Run Integration Tests
        run: make integration_test
      - name: Build application
        run: make build
      - name: Package Application
        run: make package
      - name: List dist output
```

Checkpoint 🎯

- Une image Docker fournit un environnement de système de fichier auto-suffisant (application, dépendances, binaires, etc.) comme modèle de base d'un conteneur
- On peut spécifier une recette de fabrication d'image à l'aide d'un `Dockerfile` et de la commande `docker image build`
- Les images Docker ont une convention de nommage permettant d'identifier les images très précisément

⚠ Friends don't let friends use `latest` ⚠

Versions

Pourquoi faire des versions ?

- Un changement visible d’un logiciel peut nécessiter une adaptation de ses utilisateurs
- Un humain ça s’adapte, mais un logiciel il faut l’adapter!
- Cela permet de contrôler le problème de la compatibilité entre deux logiciels.

Une petite histoire

Voi développe un client mobile qui utilise l'API exposée par le vehicle-server. Imaginez que vous changiez le contenu de la réponse de `list vehicles`

```
{
  "vehicles": [
      {
        "id": 12,
        "longitude": 34.33,
        "latitude": 43.5343,
-       "shortcode": "abef"
+       "short_code": "abef"
      }
  ],
}
```

Copy

Que se passe t'il du côté de votre application ?

💥 Plus personne ne peut dévérouiller de véhicle! 💥

Qu’est s’est il passé ?

- Le client mobile ne s’attendait pas à ce que le ShortCode du vehicule soit retourné sous l’attribut `short_code`!
- Vous avez "changé le contrat" de votre API d’une façon non rétrocompatible avec votre l’existant.
  - Cela s’appelle un 🇬🇧 **Breaking Change**

Comment éviter cela ?

- Laisser aux utilisateurs une marge de manoeuvre pour "accepter" votre changement.
  - Donner une garantie de maintien des contrats existants.
  - Informer vos utilisateurs d’un changement non rétrocompatible.
  - Anticiper les changements non rétrocompatibles à l’aide de stratégies (dépréciation).

Bonjour versions !

- Une version cristallise un contrat respecté par votre application.
- C’est un jalon dans l’histoire de votre logiciel

Quoi versionner ?

Le problème de la compatibilité existe dès qu'une dépendance entre deux bouts de code existe.

- Une API
- Une librairie
- Un langage de programmation
- Le noyau linux

Version sémantique

La norme est l'utilisation du format vX.Y.Z (Majeur.Mineur.Patch)

(source betterprograming)

Un changement **ne changeant pas le périmètre fonctionnel** incrémente le numéro de version **patch**.

Un changement changeant le périmètre fonctionnel de façon **rétrocompatible** incrémente le numéro de version **mineure**.

Un changement changeant le périmètre fonctionnel de façon **non rétrocompatible** incrémente le numéro de version **majeure**.

En résumé

- Changer de version mineure ne devrait avoir aucun d’impact sur votre code.
- Changer de version majeure peut nécessiter des adaptations.

Concrètement avec une API

- Offrir a l'utilisateur un moyen d'indiquer la version de l'API a laquelle il souhaite parler
  - Via un préfixe dans le chemin de la requête:
    - `https://vehicles.voi.com/v2.3/vehicles`
  - Via un en-tête HTTP:
    - `Accept-version: v2.3`

Version VS Git

- Un identifiant de commit est de granularité trop faible pour un l'utilisateur externe.
- Utilisation de **tags** git pour définir des versions.
- Un **tag** git est une référence sur un commit.

Nous sommes prêts, il est grand temps de faire la release de notre v1.0.

…mais c’est quoi notre production déjà?

🏝️ Notre production sera…

- Une image Docker de l'application…
- … visible sur le Docker Hub…
- … avec un (Docker) tag pour chaque version

🎓 🐳 Docker Hub

- Si vous n’avez pas déjà un compte sur le Docker Hub, créez-en un maintenant (nécessite une validation)
- Une fois authentifiés, naviguez dans votre compte (en haut à droite, "My Account")
- Allez dans la section "Security" et créez un nouvel "Access Token"
  - Permissions: "Read & Write" (pas besoin de "Delete")
  - ⚠ Conservez ce token dans un endroit sûr (ne PAS partagez à d’autres)

💡 Activer le 2FA est une bonne idée également

🎓 Taguez et déployez la version 1.0.0

- Depuis GitPod, créez un tag git local `1.0.0`
  - 💡 `git tag 1.0.0 -a -m "Première release 1.0.0, mode manuel"`
- Fabriquez l'image Docker avec le tag (Docker) 1.0.0
  - 💡 `make package` ?
- Déployez l'image sur le DockerHub
  - 💡 `docker login`, `docker image push`
- Publier le tag sur votre "remote" `origin`.
  - 💡 `git push origin 1.0.0`
- 💡 Peut être faire un `make release` qui englobe git tag et docker push ?

✅ "Taguez" et déployez la version 1.0.0

```
.PHONY all
all: clean dist unit_test integration_test build package release

.PHONY: release
release:
  git tag $(TAG) -m "$(TAG_MESSAGE)"
  git push $(TAG)
  docker push $(IMAGE):$(TAG)
```

Copy

```
docker login --username=<VOTRE USERNAME>

make all TAG="1.0.0" "TAG_MESSAGE="Première Release, Version Manuelle"
```

Copy

Vérifiez Git et DockerHub après ça!

Checkpoint 🎯

- La notion de "version" est un outil de communication aux consommateurs de nos produits logiciels
- Le "semantic versioning" est une des façon les plus usitées pour gérer les politiques de version
- Nous avons déployé manuellement notre première image Docker, avec synchronisation code source <→ image Docker

⇒ 🤔 C'était très manuel. Et si on regardait à automatiser tout ça ?

"Continuous Everything"

Livraison Continue

Continuous Delivery (CD)

🤔 Pourquoi la Livraison Continue ?

- Diminuer les risque liés au déploiement
- Permettre de récolter des retours utilisateurs plus souvent
- Rendre l'avancement visible par **tous**

> *How long would it take to your organization to deploy a change that involves just one single line of code?*
>
> — Mary and Tom Poppendieck

Qu’est ce que la Livraison Continue ?

- Suite logique de l’intégration continue:
  - Chaque changement est **potentiellement** déployable en production
  - Le déploiement peut donc être effectué à **tout** moment

> *Your team prioritizes keeping the software **deployable** over working on new features*
>
> — Martin Fowler

La livraison continue est l'exercice de **mettre à disposition automatiquement** le produit logiciel pour qu'il soit prêt à être déployé à tout moment.

Livraison Continue avec GitHub Actions

Prérequis: exécution conditionnelle des jobs

Il est possible d'exécuter conditionnellement un job ou un step à l'aide du mot clé `if` (documentation de `if`)

```yaml
jobs:
  release:
    steps:
      # Lance le step dire coucou uniquement si la branche est main.
      - name: "Dire Coucou"
        run: echo "coucou"
        if: contains('refs/heads/main', github.ref)
```

Copy

🎓 Secret GitHub / DockerHub Token

- Reprenez (ou recréez) votre token DockerHub
  - 💡 Documentation "Manage access tokens"
- Insérez le token DockerHub comme secret dans votre dépôt GitHub
  - 💡 Creating encrypted secrets for a repository

🎓 Livraison Continue sur le DockerHub

- **But :** Automatiser le déploiement de l'image dans le DockerHub lorsqu'un tag est poussé
- Changez votre workflow de CI de façon à ce que, sur un push de tag, les tâches suivantes soient effectuées :
  - Comme avant: Build, Tests, Build Package
  - SI c'est un tag, alors il faut pousser (et éventuellement reconstruire avec le bon nom) l'image sur le DockerHub
- 💡 Il va falloir adapter `make release` pour qu'il ne pousse plus de tag
- 💡 Utilisez les GitHub Action suivantes :

✅ Livraison Continue sur le DockerHub

Copy

```yaml
# ...
    steps:
      # ... make unit_test
      # ... make integration_test
      # ... make build
      # ... make package
      - name: Login to Docker Hub
        uses: docker/login-action@v3
        if: startsWith(github.ref, 'refs/tags/')
        with:
          username: xxxxx
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Push if triggered by a tag
        if: startsWith(github.ref, 'refs/tags/')
        run: |
          make release  TAG="${{github.ref_name}}"
```

Déploiement Continu

🇬🇧 Continuous Deployment / "CD"

🤔 Qu’est ce que le Déploiement Continu ?

- Version "avancée" de la livraison continue:
  - Chaque changement **est** déployé en production, de manière **automatique**

Continuous Delivery versus Deployment

Source : http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Bénéfices du Déploiement Continu

- Rends triviale les procédures de mise en production et de rollback
  - Encourage à mettre en production le plus souvent possible
  - Encourage à faire des mises en production incrémentales
- Limite les risques d'erreur lors de la mise en production
- Fonctionne de 1 à 1000 serveurs et plus encore…

🎓 Déploiement Continu sur le DockerHub

- **But :** Déployer votre image `vehicle-server` continuellement sur le DockerHub
- Changez votre workflow de CI de façon à ce que, sur un push sur la branch `main`, les tâches suivantes soient effectuées :
  - Comme avant: on joue le cycle de vie via make.
  - SI c'est la branche `main`, alors il faut pousser l'image avec le tag `main` sur le DockerHub
  - Conservez les autre cas avec les tags

✅ Déploiement Continu sur le DockerHub

Copy

```yaml
# ...
    steps:
      # ... make unit_test
      # ... make integration_test
      # ... make build
      # ... make package
      # ... Tag release!
      - name: Login to Docker Hub
        uses: docker/login-action@v3
        if: contains('refs/heads/main', github.ref)
        with:
          username: xxxxx
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Push if on `main` branch
        if: contains('refs/heads/main', github.ref)
        run: |
          make release  TAG="${{github.ref_name}}"
```

Checkpoint 🎯

- La livraison continue et le déploiement continu étendent les concepts du CI
- Les 2 sont automatisées, mais un être humain est nécessaire comme déclencheur pour la 1ère
- Le choix dépends des risques et de la "production"
- On a vu comment automatiser le déploiement dans GitHub Actions
  - Conditions dans le workflow
  - Gestion de secrets

Évaluation du Projet

- L’évaluation pour le cours de CI/CD sera sur votre projet final
- Cela reste une note indépendante /20

Barème (indicatif):

- Utilisation de Git et GitHub (/5)
  - Est-ce que le Git Flow est clair?
  - Est-ce qu'il y a des PRs? Bien découpées? De la revue de code?
  - Est-ce qu'il y à des tags et des version sémantiques?

- Cycle de vie (/3)
  - Est-ce que les différentes étapes de votre livraison sont claires?
  - Est-ce qu'un outil les normalise?
- Gestion des Dépendances (/2)
  - Est-ce que vos dépendances sont récupérées de façon reproductible?

- Intégration Continue (/5)
  - Est-ce qu'il y à un outil de CI?
  - Que prouve votre CI?

- Livraison Continue (/3)
  - Est-ce que vous avez un processus de livraison clair?
  - Est-ce automatisé?
  - Est-ce que votre livrable est utilisable? Documenté?
- Tests (/2)
  - Avez vous une suite de tests automatisée? Que prouve t'elle?

BONUS TRACK

- git rebase (avec -i ❤)
- git commit --amend
- git add -p
- git revert
- git reflog
- OH SHIT GIT

Dependabot!

- Ajoutez la mise à jour Automatique des Dépendances avec Dépendabot Docs

Lint du Projet

- Ajoutez du lint automatique de code go avec l'action Golang CI lint

AMA (Ask Me About)

- Infrastructure & Cloud?
- Monitoring et Observabilité?
- Open Source?
- Career ? Remote Work?
- …?

Bibliographie

Ligne de commande

- https://blog.balthazar-rouberol.com/category/essential-tools-and-practices-for-the-aspiring-software-developer
- https://tldp.org
- https://en.wikipedia.org/wiki/POSIX
- https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
- https://linuxhandbook.com/linux-directory-structure/

Git / VCS

- https://docs.github.com
- https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
- http://martinfowler.com/bliki/VersionControlTools.html
- http://martinfowler.com/bliki/FeatureBranch.html
- https://about.gitlab.com/2014/09/29/gitlab-flow/
- https://www.atlassian.com/git/tutorials/comparing-workflows
- http://nvie.com/posts/a-successful-git-branching-model/

Intégration Continue

- http://martinfowler.com/articles/continuousIntegration.html
- http://martinfowler.com/bliki/ContinuousDelivery.html
- https://jaxenter.com/implementing-continuous-delivery-117916.html
- https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/
- http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model
- http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Livraison/Déploiement Continu

- http://martinfowler.com/articles/continuousIntegration.html
- http://martinfowler.com/bliki/ContinuousDelivery.html
- https://jaxenter.com/implementing-continuous-delivery-117916.html
- https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/
- http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model
- http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Tests

- (FR) http://douche.name/blog/nomenclature-des-tests-logiciels/
- http://martinfowler.com/bliki/UnitTest.html
- https://en.wikipedia.org/wiki/Software_testing
- http://martinfowler.com/tags/testing.html
- http://martinfowler.com/bliki/TestCoverage.html
- http://martinfowler.com/bliki/TestDrivenDevelopment.html

Autre

- https://dduportal.github.io/cours/
- https://www.jenkins.io/node/

Merci !

- damien.duportal <chez> gmail.com
- @DamienDuportal
- jlevesy <chez> gmail.com
- @jlevesy

Slides: https://cicd-lectures.github.io/slides/2024