

Introduction au CI/CD

ENSG - Décembre 2023

- Présentation disponible à l'adresse: <https://cicd-lectures.github.io/slides/2023>
- 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
 - N'hésitez pas ouvrir des Pull Request si vous voyez des améliorations ou problèmes: sur cette page (😉 wink wink)



Une petite histoire du génie logiciel



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
- Un compte sur Google
- 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 :

User Validation Required

⚠ To use Gitpod you'll need to validate your account with your phone number. This is required to discourage and reduce abuse on Gitpod infrastructure.

Enter a mobile phone number you would like to use to verify your account. Having trouble? [Contact support](#)

Mobile Phone Number

Send Code via SMS

▲ 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⚠

The screenshot shows the GitPod Workspaces interface. At the top, there's a header with the GitPod logo, a 'Workspaces' button, a 'New Team' button, and a 'Feedback' button. Below the header, the title 'Workspaces' is displayed, followed by the sub-instruction 'Manage recent and stopped workspaces.' A search bar labeled 'Filter Workspaces' and a limit selector 'Limit: 50' are present. A green button for 'New Workspace' is also visible. The main area lists two workspaces:

Workspace Name	Repository	Branch	Last Update	More Options	
cicdlectures-gitpod-jcu32jcv4q2	github.com/cicd-lectures/gitpod	cicd-lectures/gitpod - main	https://github.com/cicd-lectures/gitpod	main No Changes 3 minutes ago	⋮
cicdlectures-gitpod-vv8g7mywidp	github.com/cicd-lectures/gitpod	cicd-lectures/gitpod - main	https://github.com/cicd-lectures/gitpod	main No Changes 4 minutes ago	⋮



Permissions GitPod <→ GitHub



- Pour les besoins de ce cours, vous devez autoriser GitPod à pouvoir effectuer certaines modifications 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 droite) 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>

5 . 10

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 commandes

- Afficher le manuel de <commande> :

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

Copy

- 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, chercher 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 process 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

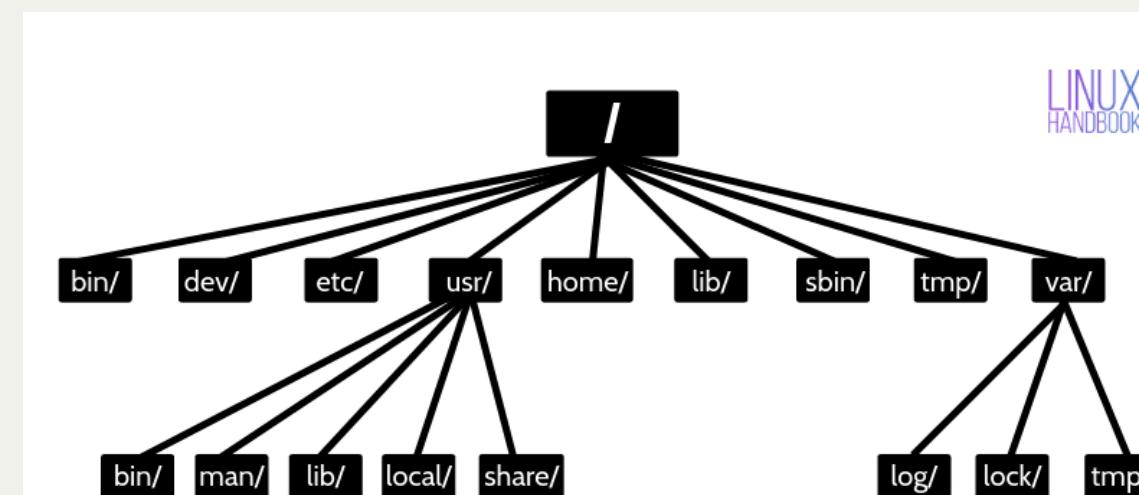


Commandes de base 2/2

- 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}

# Recommandation
echo "${MA_VARIABLE}"

MA_VARIABLE="Salut tout le monde"

echo "${MA_VARIABLE}"
```

Copy

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

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

Copy



- 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 $?

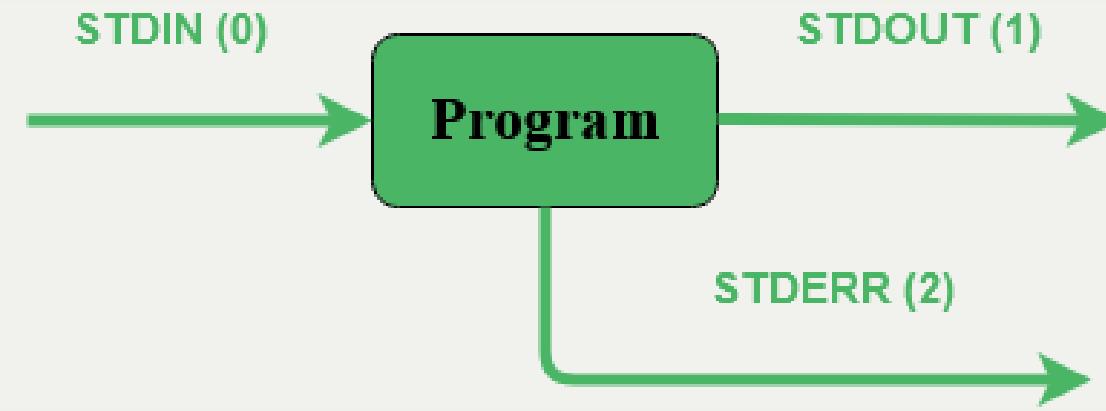
ls /do_not_exist
echo $?

# 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 fichiers 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 entree?



1. Resolution DNS
2. Connection TCP (eventuellement TLS)
3. Envoi d'une requette HTTP
4. Reception d'une reponse
5. Rendu de la page.



Zoom sur HTTP

Faire l'anatomie d'une requete / reponse HTTP, Verbes, Headers, Body, Status code etc...

Expliquer que le but c'est de normaliser un format d'echange de donnees.



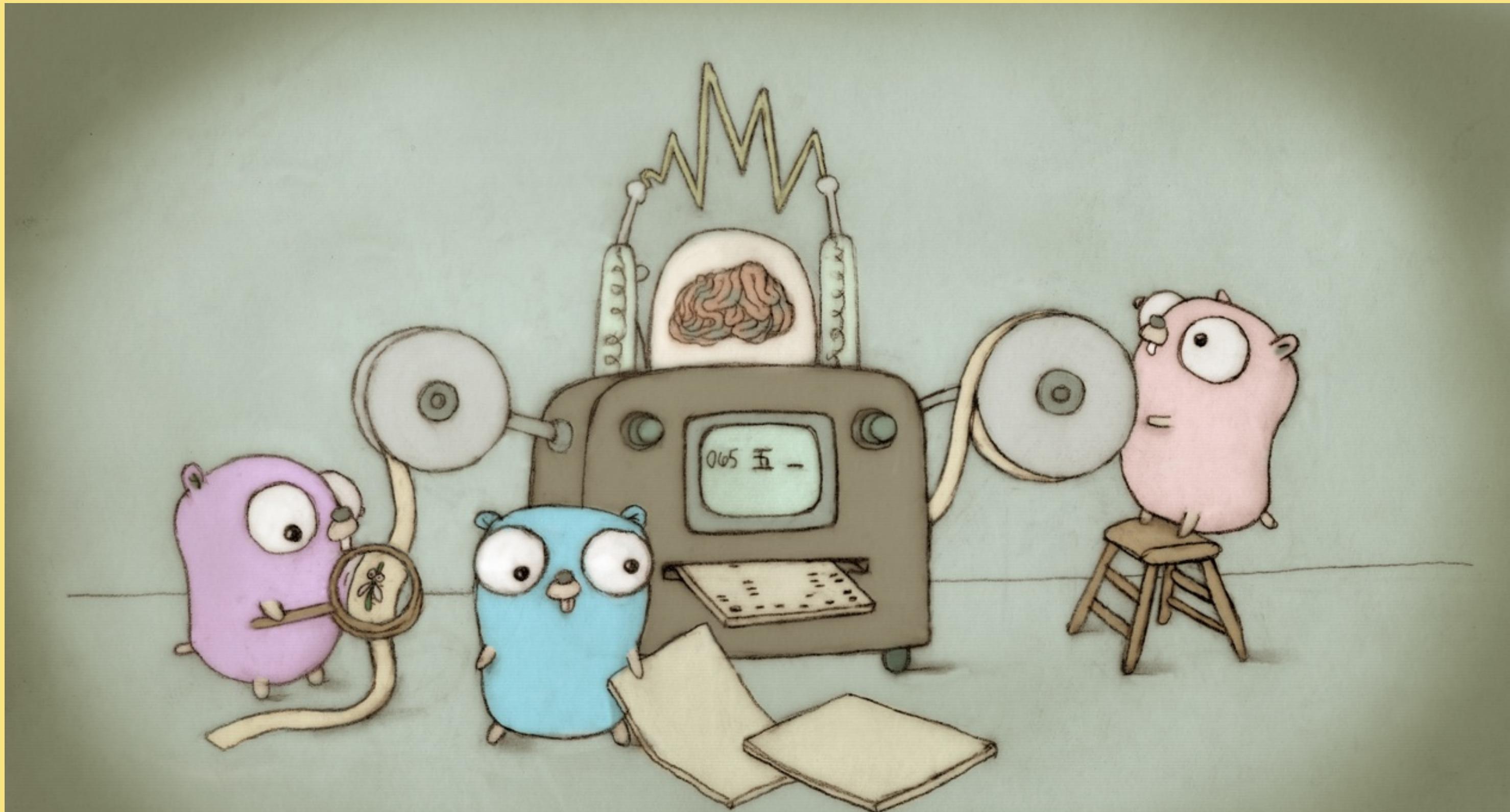
Comment parler HTTP depuis le terminal?

Présentation de curl. donner des exemples pratiques pour utiliser -v, -L, --data, -X. Poster du contenu sera vers l'API.

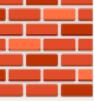
Exercice

Les faire curler voi.com, les faire observer toutes les etapes de la requete, la 301 upgrade vers https avec -v et -o /dev/null





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

```
package main

import (
    "fmt"
)

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

Copy

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



✓ Solution: Un Premier Programme en Go

```
# 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
```

Copy



go run ./main.go compile et execute le programme directement!

```
$ 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 😊

Anatomie d'un Fichier go (1/2)

```
package main
```

```
import (
    "fmt"
)

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

Copy



Anatomie d'un Fichier go (2/2)

- func: Declare une fonction main:
 - Cette fonction appelle la fonction `Println` du package `fmt`
 - En passant la chaîne de caractères "Hello ENSG!"
- package: declare 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 où le télécharger, comme ça tous les packages sont uniques!
 - Ex: `github.com/jlevesy/prometheus-elector/config`
- Sauf pour la librairie standard, où 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 presents dans un même repertoire doivent declarer le meme package
- **Convention:** Le nom du package déclaré est le même que celui du répertoire content le fichier
 - Ex: `github.com/jlevesy/prometheus-selector/config` ➡ 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 appellée 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.

```
// fonction privée du package courant. Ne peut pas être utilisée à 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 appellée depuis un autre package.
func PublicFunc() {
    // appelle la fonction `OpenFile` du package os.
    // ✅ compile, `OpenFile` est exportée!
    file, _ := os.OpenFile("/some/file")
}
```

Copy

?

8.13

Controller 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

```
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

```
func main() {  
    // Declare 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!" à la variable message.  
    message = "Hello ENSG!"  
  
    // Est équivalent à:  
    var message string = "Hello ENSG!"  
  
    // Est encore équivalent a... 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)  
}
```

Copy



Variables (2/3)

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

```
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
}
```

Copy



Variables (3/3)

⚠ Go est un langage fortement typé ⚠

```
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



Types Scalaires

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

Controle de Flot



Controle de Flot: if (1/2)

```
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
}
```

Copy



Controle de Flot: if (2/2)

```
func main() {
    // if / else avec short statement.
    // avantage: ok n'exsite 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
}
```

Copy



Controle de Flot: switch

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

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

Copy



Controle de Flot: boucle for

```
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 🎉

```
// 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  
}
```

Copy

?

8.25

Fonctions(2/4)

- Les fonctions peuvent être manipulées comme des valeurs!

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

    return fn
}
```

Copy



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

```
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

?

8.27

Fonctions(4/4)

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

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

    addGreeting(name)

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

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

Copy



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 permet de **référencer** cet emplacement memoire en copiant son adresse dans une autre variable avec l'opérateur &. Autrement dit, on **crée un pointeur**.

```
func main() {  
    // On déclare et initialise une variable. Cela aloue de la mémoire sur la pile.  
    var message string = "Hello ENSG!"  
    // On copie l'adresse memoire 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)  
}
```

Copy

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.

```
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.

```
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

```
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 chaîne de caractères "Hello " concaténée avec
    // la valeur de la variable référencée par namePtr.
    *namePtr = "Hello " + *namePtr
}
```

Copy



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 à nil.
 - ...Sinon il faudra la gérer!

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

Copy

Gestion d'Erreur (2/2)

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



Collections



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

```
func main() {  
    // Declare et initialise un tableau de 2 strings.  
    var intArray [2]string  
    // On peut assigner un élément du tableau en utilisant son index.  
    intArray[0] = 1  
    intArray[1] = 3  
    // On accède à un élément du tableau en utilisant son index.  
    fmt.Println(intArray[0], intArray[1])  
  
    anotherArray := [4]int{2, 4, 6, 8}  
    // X Ne compile pas: la taille fait partie du type!  
    // On assigne un tableau de 4 entrées à un tableau de deux entrées  
    intArray = anotherArray  
}
```

Copy



Collections: Slices (1/5)

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

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

    // Declare et initialise une slice référençant les entrées
    // entre l'index 1 et 3 du tableau anArray.
    // Se lit interval [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]
}
```

Copy



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`

```
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 à l'aide de la fonction `len`
- Sa capacité: la taille totale du tableau référencé
 - On y accède à l'aide de la fonction `cap`

```
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

- On peut concaténer des objets à une slice avec l'opérateur append
- Δ Cela n'ajoute pas nécessairement un entrée à la slice passée en paramètre.
 - Dans le cas où le tableau sous jacent est plein ($\text{len} == \text{cap}$), append va réallouer un tableau et copier toutes les entées dans ce nouveau tableau.
 -  En conséquence: il faut **TOUJOURS** assigner la valeur renournée par append

```
func main() {  
    // On ajoute l'entrée 10 à 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` à 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 un arrêt de l'exécution
- En revanche: `append` et `len` savent gérer une `nil` slice.

```
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

Parcourir une slice ou un tableau

- 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

```
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](#)

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

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

func toStringSlice(input []int) []string {
    // On alloue une slice de string de la taille de la slice d'entiers donnée en paramètre.
    result := make([]string, len(input))

    // Pour chaque entrée de la slice input...
    for i, v := range input {
        // On écrit le résultat de la conversion
        // dans la slice de résultat à l'index courant.
        result[i] = strconv.Itoa(v)
    }

    return result
}
```

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 ou les slices
 - Une map peut être nil, `len` retournera 0.



Collections: maps (2/3)

Exemple d'écriture

```
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 supprimer une entrée d'une map
    delete(mapVilles, "Julien")

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

Copy

Exemple de lecture

```
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, retourne 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 à true  
    // Si la clé n'existe pas, ok sera false.  
    ageMichel, ok := mapAges["Michel"]  
    if !ok {  
        fmt.Println("Pas d'âge pour Michel")  
    } else {  
        fmt.Println("Age de Michel", ageMichel)  
    }  
}
```

Copy

Parcourir une map

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

```
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



Exercice: Comptez les occurrences de mots dans une chaîne de caractère

- Ecrivez une fonction WordCount en go qui accepte une chaîne de caractère et qui retourne le nombre d'occurrences de chacun des mots contenu dans la chaîne.
 - **Indice:** La signature de votre fonction devrait ressembler à `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 chaîne de caractère et retourne une `string`.

✓ Solution: Comptez les occurrences de mots dans une chaîne de caractère

```
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
}
```

Copy



- 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.

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

func main() {
    // On déclare 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 >>
}
```

Copy

Structures (2/3)

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

```
// 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")  
    }  
}
```

Copy

- 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.

```
// 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() {
```

Copy

?

8.54

- Nous avons utilisé le mot clef `type` pour définir un nouveau type de structure.
- Mais `type` peut être appliqué à des tas d'autre choses.

```
// 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

- Il est possible d'attacher des méthodes aux types que l'on définit

```
// Définit un type Color représenté en mémoire par un entier.  
type Color int  
  
// Attache une méthode String à toute instance de la valeur Color  
// qui retourne le nom de la couleur sous forme de chaîne 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 (  
    ColorBlue Color = 1  
    ColorGreen Color = 2  
)  
?
```

Copy

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.

```
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()
}
```

Copy

?

8.57

Receveurs: Pointeurs ou Valeurs?

- On peut attacher une méthode sur une valeur du type, ou sur un pointeur.
- L'opérateur . (accès) référence et déréférence les pointeurs implicitement.
- Quelques règles:
 - Sur les types scalaires (int, string etc...) on préfèrera les valeurs, la copie ne couture rien.
 - Sur les collections et fonctions, on garde les valeurs, ce sont des types références.
 - Sur les structs, on préfère attacher au pointeurs car cela évite une copie parfois lourde.



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**.



```
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".
```

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

```
// 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

?

8.62

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

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`
 - Receveurs de méthodes (ie `*Store ⇒ st`)
 - Et d'autres...



Exemples concrets

- Decoder du JSON
- Faire une requete HTTP.



Exercice:

Ecrire un programme qui fait une requete http vers une API en JSON, definir un type structure decoder le payload et affiche un resultat.

<https://swapi.dev/> par exemple, on leur fait chercher le diametre de Tatooine en indiquant que tatooine c'est ID 1

Si ils sont chaud on leur fait chercher la cargo capacity de la death star, sans leur donner l'ID du coup ils vont devoir fait un appel a index

Références

- <https://www.youtube.com/watch?v=xi8732QO33Y>
- <https://go.dev/tour/>
- <https://gobyexample.com/>
- https://go.dev/doc/effective_go
- <https://go.dev/doc/>
- <https://go-proverbs.github.io/>
- <https://go.dev/play/>



Les fondamentaux de git



Tracer le changement dans le code

avec un **VCS** :  Version Control System

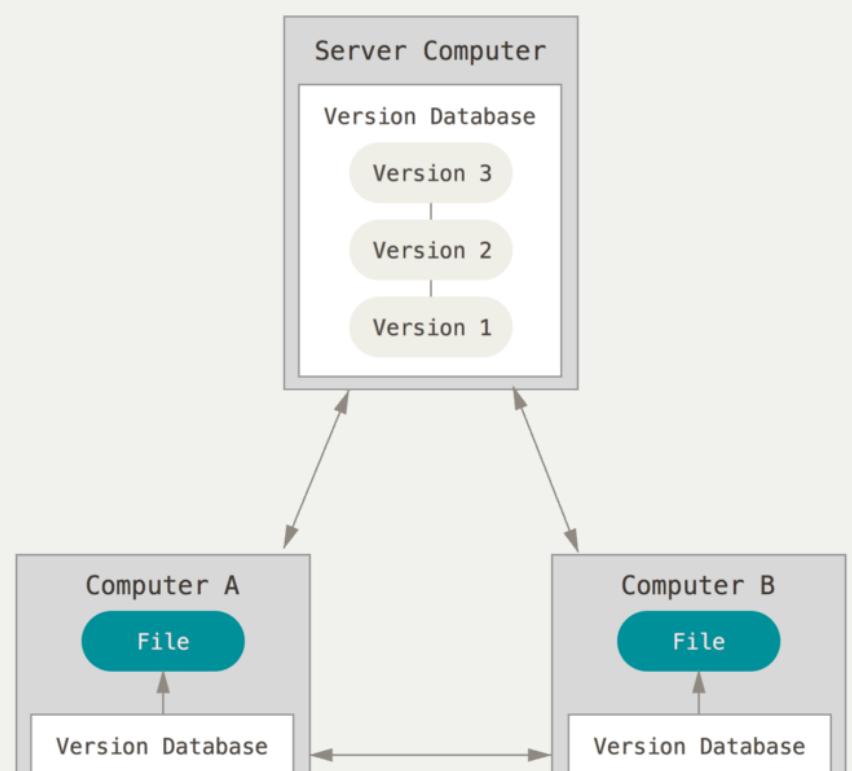
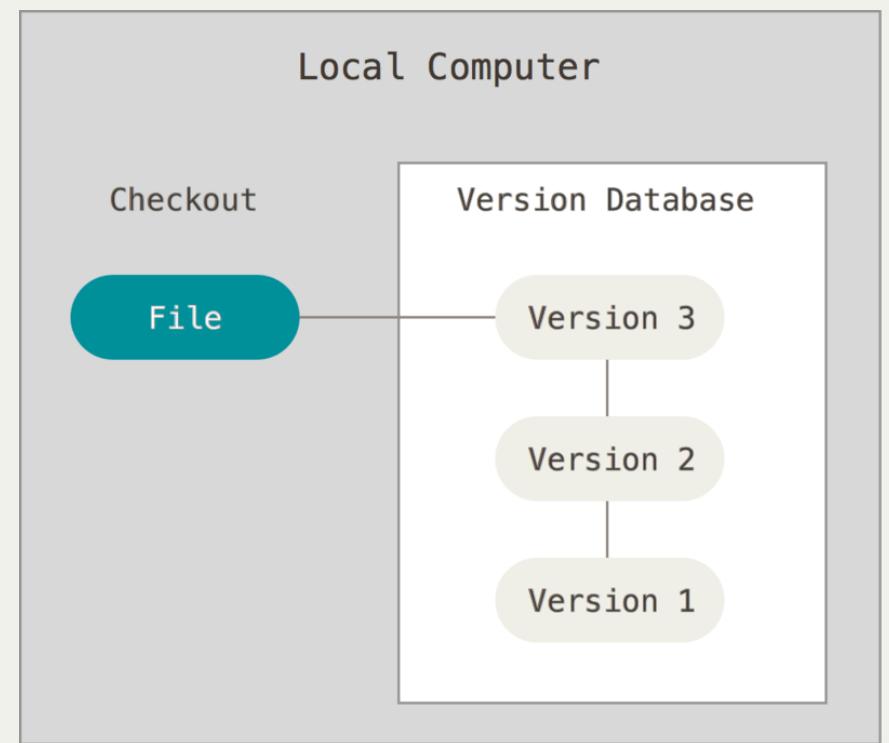
également connu sous le nom de SCM ( Source Code Management)



Pourquoi un VCS ?

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





Quel VCS utiliser ?



Nous allons utiliser Git

Git

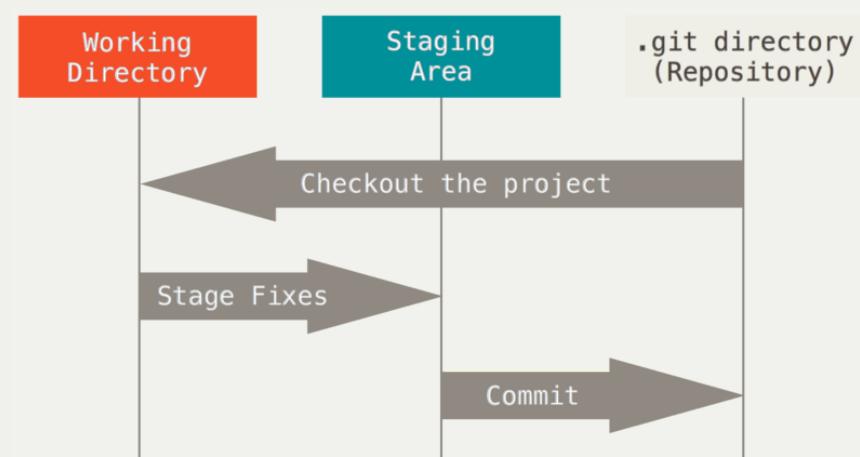
Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

<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 committted"
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

```
v 10 [REDACTED] cluster/addons/node-problem-detector/npd.yaml ...
@@ -26,28 +26,28 @@ subjects:
26   apiVersion: apps/v1
27   kind: DaemonSet
28   metadata:
29   - name: npd-v0.8.0
30     namespace: kube-system
31     labels:
32       k8s-app: node-problem-detector
33   - version: v0.8.0
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
41     template:
42       metadata:
43         labels:
44           k8s-app: node-problem-detector
45   - version: v0.8.0
46     kubernetes.io/cluster-service: "true"

26   apiVersion: apps/v1
27   kind: DaemonSet
28   metadata:
29   + name: npd-v0.8.5
30     namespace: kube-system
31     labels:
32       k8s-app: node-problem-detector
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: v0.8.5
41     template:
42       metadata:
43         labels:
44           k8s-app: node-problem-detector
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.

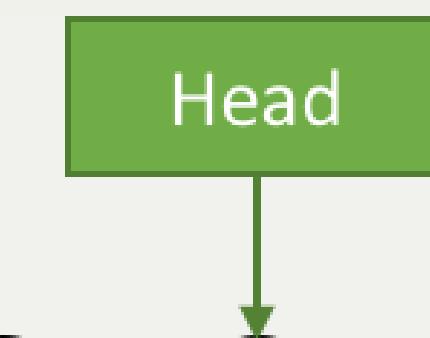
- > 3 [REDACTED] Jenkinsfile
- > 10 [REDACTED] make.ps1
- > 456 [REDACTED] tests/plugins-cli.Tests.ps1

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

A screenshot of a GitHub commit page. At the top, there is a green checkmark icon followed by the text "Bump node-problem-detector to v0.8.5". Below this is a "Browse files" button. Underneath the commit message, it says "master (#96716)". To the left is a user icon and the name "tos13k committed 2 days ago". To the right, it shows "1 parent e64ebe0 commit 8f2dd3aaab02c3b4c1c8233aa5f93bb439f23228". At the bottom left, it says "Showing 3 changed files with 8 additions and 8 deletions." and at the bottom right, there are "Unified" and "Split" buttons.

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

O : a commit





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

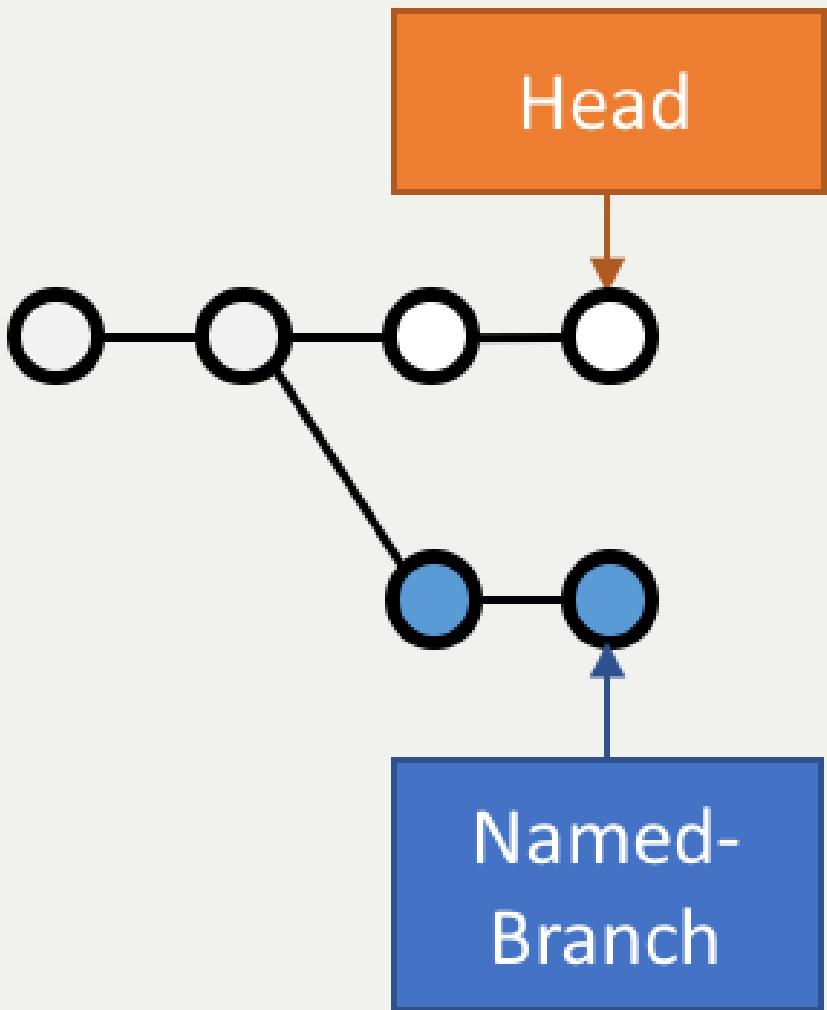
git checkout -- README.md
git status
```

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

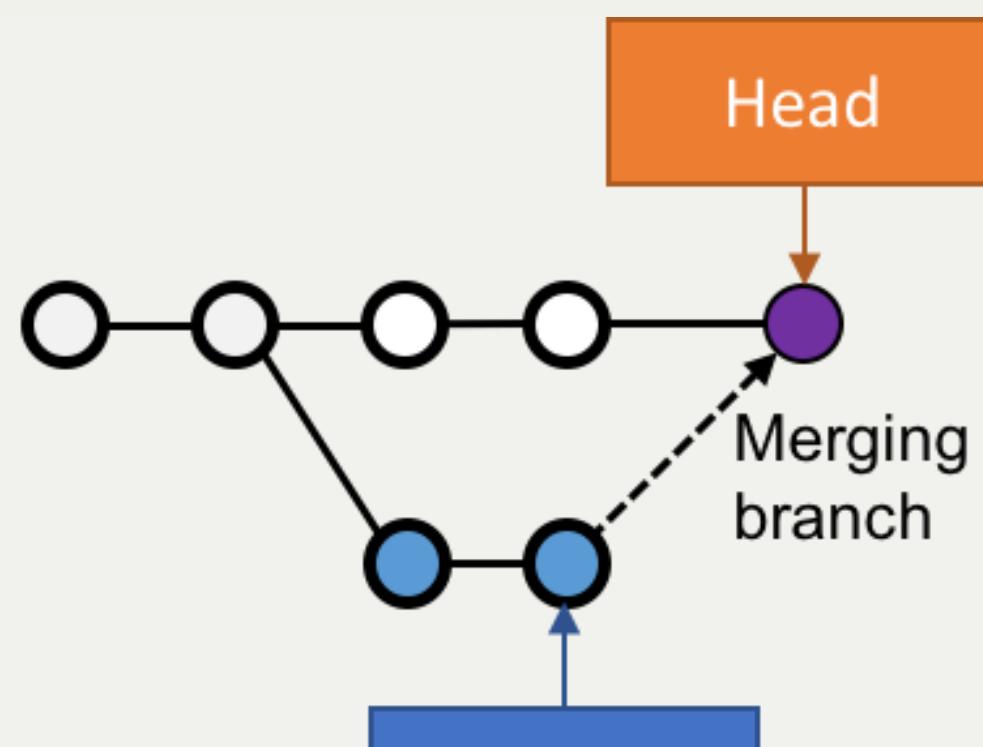
```
git branch feature/html && git switch feature/html
# Ou git switch --create feature/html
echo '<h1>Hello</h1>' > ./index.html
git add ./index.html && git commit --message="Ajout d'une page HTML par défaut" # -m / --message

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 stratégies sont possibles pour merger:
 - Quand l'historique de commit n'a pas divergé: 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, changeset, 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 (localemement)

Présentation de votre projet

TODO REFAIRE TOUTE CETTE PARTIE.



Contexte

Expliquer l'app Expliquer la fonctionnalite qu'on implemente Trouver une excuse pour justifier le fait qu'on donne une tarball.



Prise en Main du Projet (1/2)

- Une équipe technique de **Bananes** avait commencé l'implémentation du serveur, et à fourni une archive téléchargeable **ICI**, contenant le code source du projet
- **Bananes** vous assure qu'ils ont suivi toutes les "best practices" du développement logiciel sur minitel
 - Il y à un `LISEZMOI.txt` à la racine du projet :tada:
 - ... et pas grand chose d'autre?



Prise en Main du Projet (2/2)

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

# Téléchargez le projet sur votre environnement de développement
curl -sSLO https://cicd-lectures.github.io/slides/2023/media/menu-server.tar.gz

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

Copy

A partir de là vous pouvez ouvrir le fichier LISEZMOI.txt 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, tous les menus retournés s'appellent "TODO" :sob:
- Le correctif ne semble pas compliqué à faire...
- ... sauf que vous ne pouvez pas compiler le projet!

Il va falloir remédier à ça d'une façon ou d'une autre, sinon vous n'allez pas aller bien loin!





Exercice : Initialisez un dépôt git

- Nettoyez le contenu superflu du projet et initialisez un dépôt git dans le répertoire, puis créez un premier commit
- Par contenu superflu, nous entendons:
 - Tout ce qui est potentiellement généré
 - Les scripts de lancement obsolètes et inutiles
 - Un petit renommage du LISEZMOI.txt en README.md et un coup de nettoyage de son contenu



Pour chaque "popup" de l'éditeur (en bas à droite), choisissez "Oui" ou "Reload"

✓ Solution Exercice

```
# On évacue le contenu inutile
rm -rf dist/ menu-server.tar.gz
rm executer.sh
# On renomme LISEZMOI.txt en README.md
mv LISEZMOI.txt README.md
# On nettoie son contenu
code README.md

# On initialise un nouveau dépôt git
git init

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

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

Copy



Checkpoint



- Vous avez récupéré un projet Java 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*

Comment fabriquer du code?

TODO (refaire)

- Faire suivre un cycle de vie précis
 - build, lint, test, integration test, release.
- go n'impose pas forcement de cycle de vie, c'est spécifique à l'application
- On utilise Make pour normaliser le cycle de vie



TODO DETERER LES slides make?



Comment garantir que le processus de livraison est reproductible?

- Gestion des dependances
- Gestion des outils

Parler des go modules, de ce que fait la toolchain go <https://go.dev/ref/mod>, faire un resume """"digeste"""" de ca.

Mettre son code en sécurité

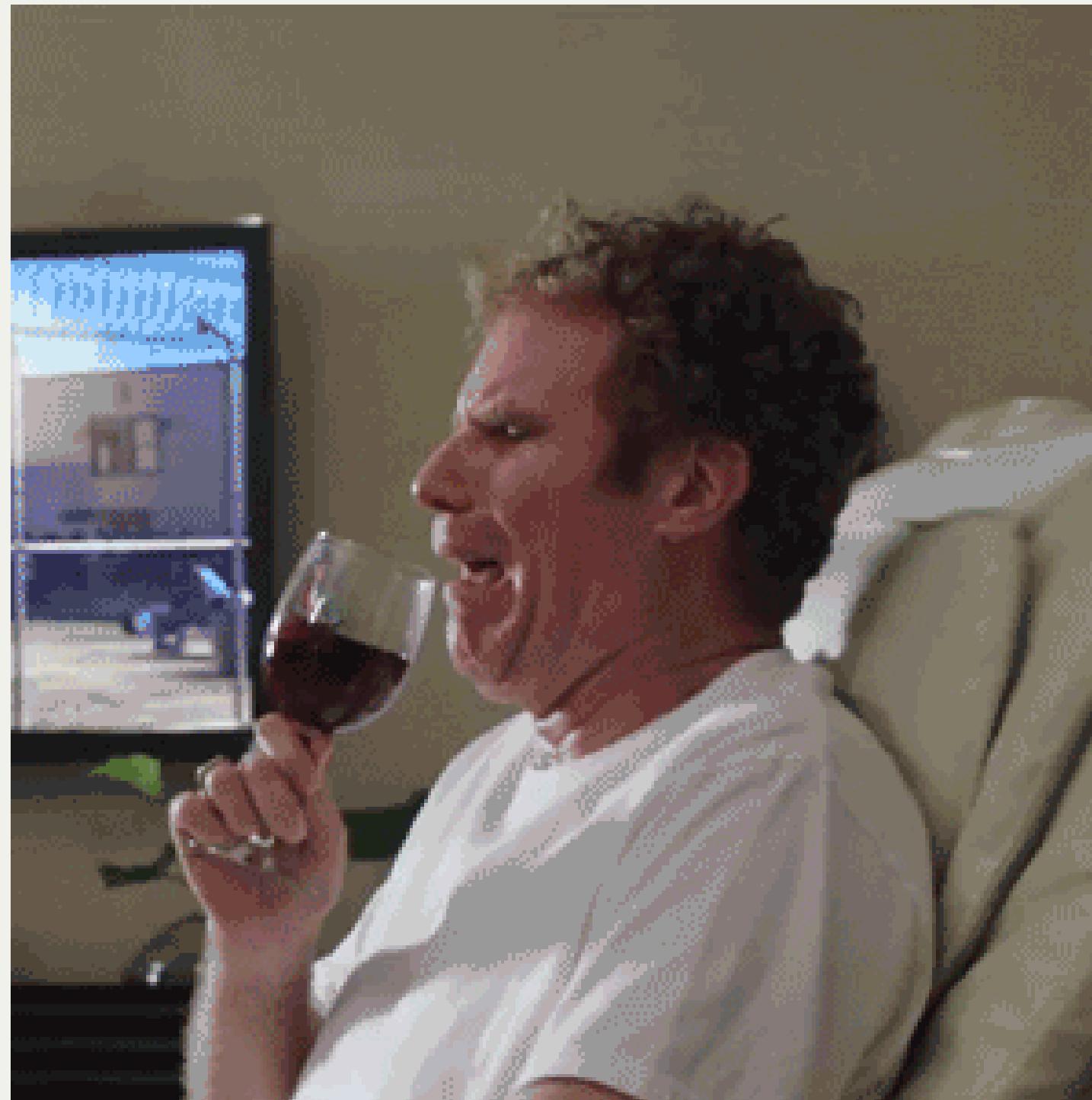


Une autre petite histoire

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 ?

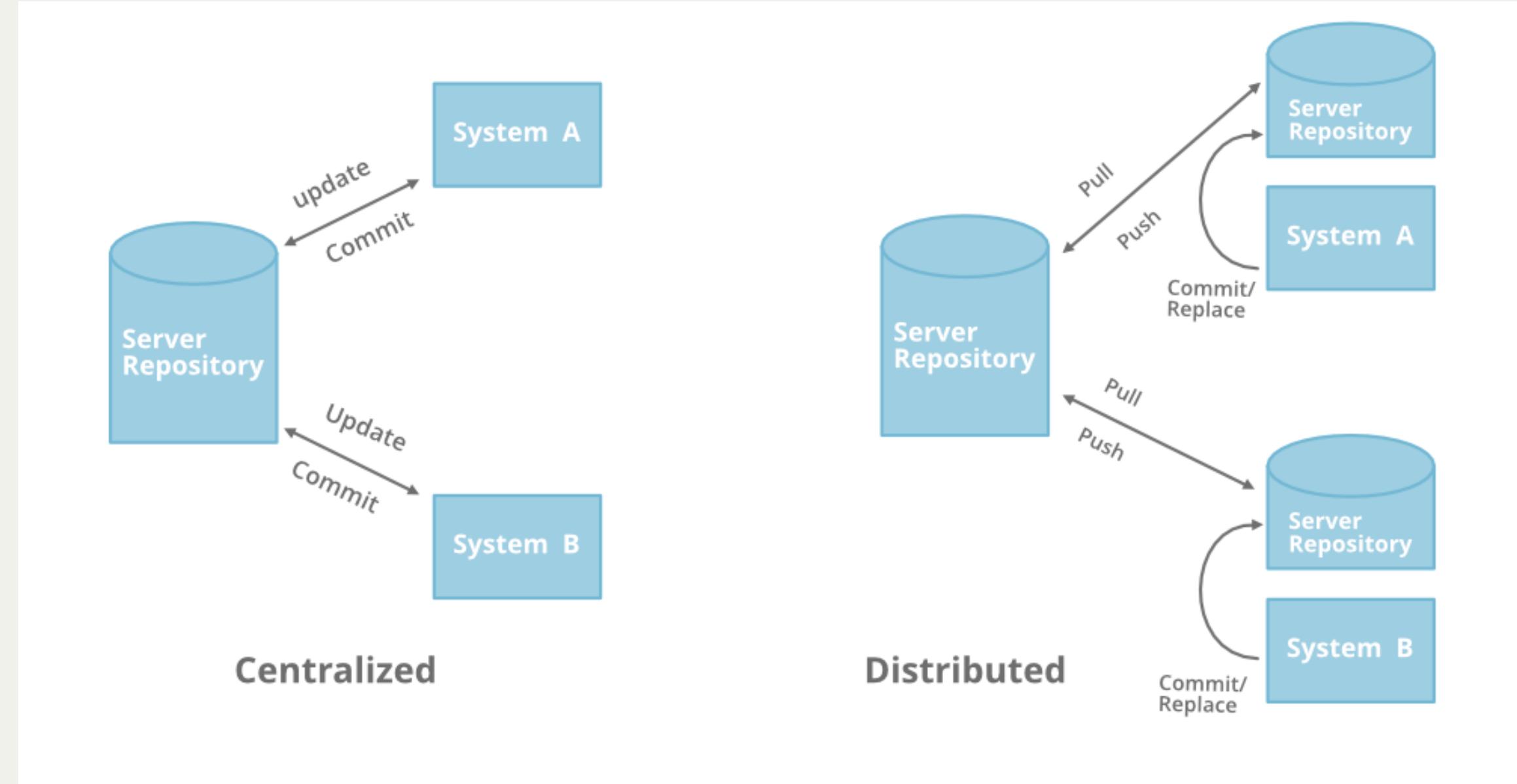
- Répliquer votre dépôt sur une ou plusieurs machines !
- Git est pensé pour gérer ce de problème



Gestion de version décentralisée

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

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 menu-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 à 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 à la bonne adresse
git remote -v
```

Copy



Publier une branche dans sur dépôt distant

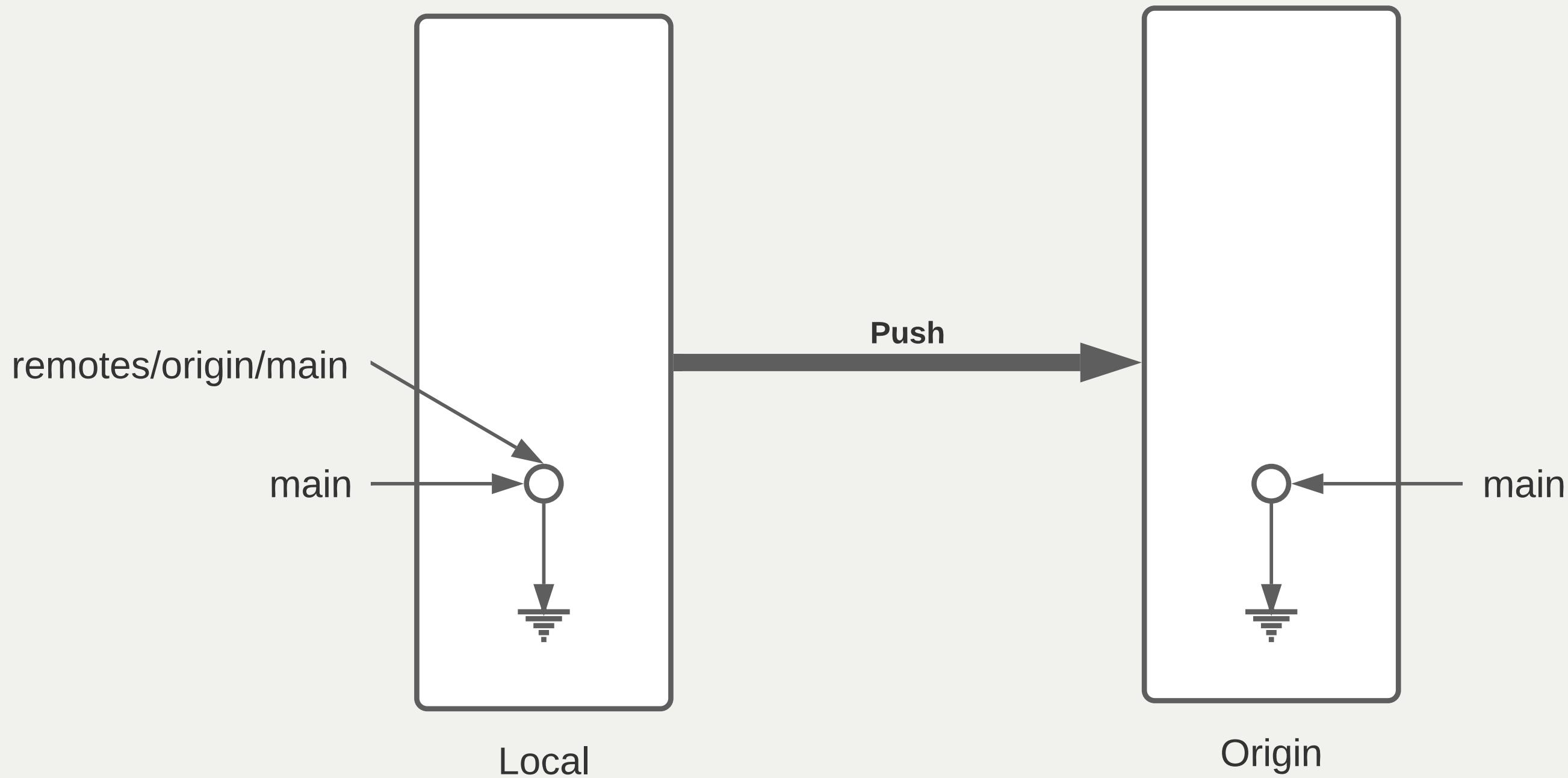
Maintenant qu'on a un dépôt, il faut publier notre code dessus !

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

Copy



Que s'est-il passé ?



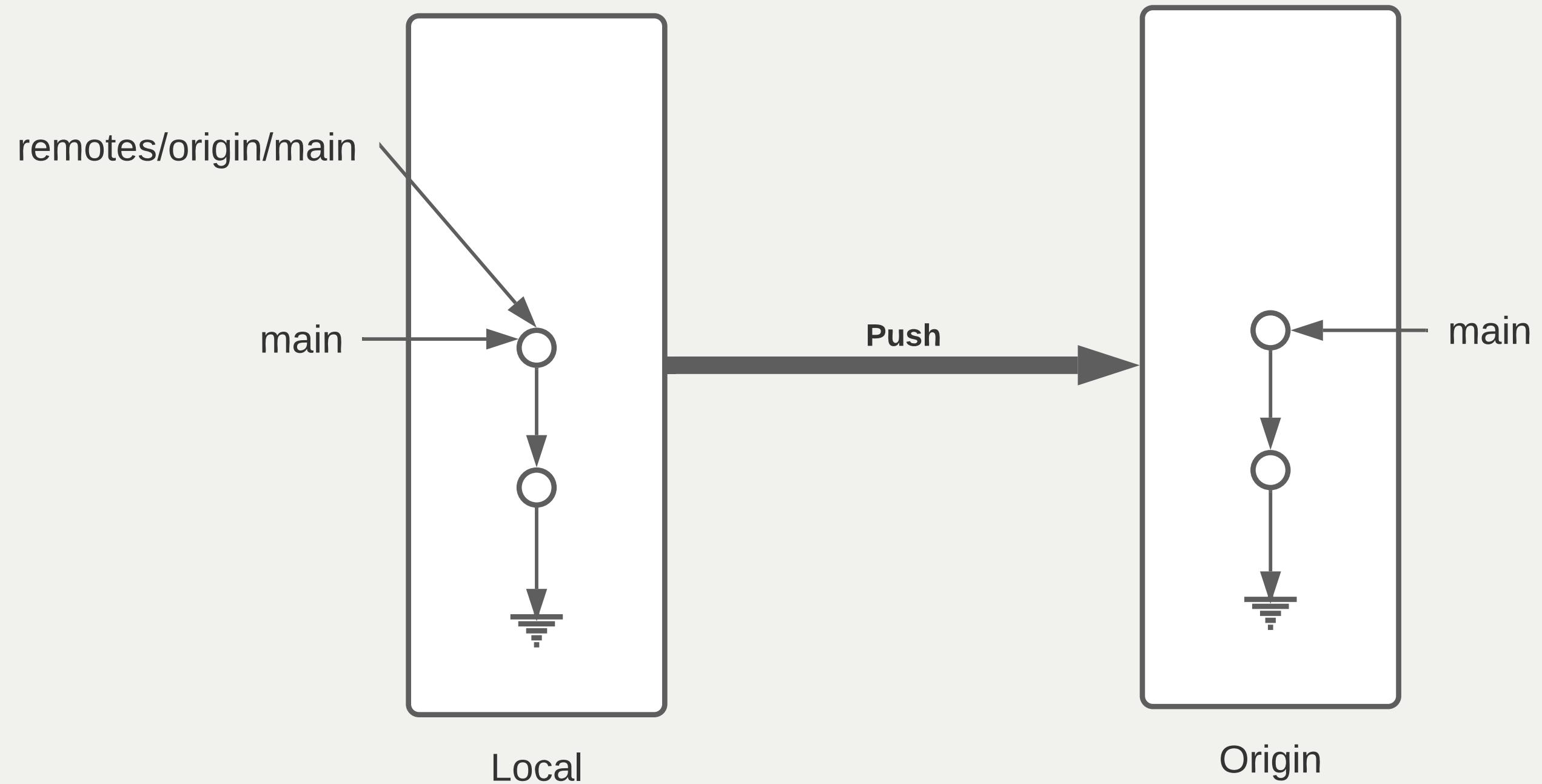
- git a envoyé la branche main sur le remote origin
- ... qui à accepté le changement et mis à jour sa propre branche main.
- git a créé localement une branche distante origin/main qui suis l'état de main sur le remote.
- Vous pouvez constater que la page github de votre dépôt affiche le code source

Refaisons un commit !

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

Copy





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>
```

Copy

```
# Lister toutes les branches y compris les branches distantes  
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!
```



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 à jour

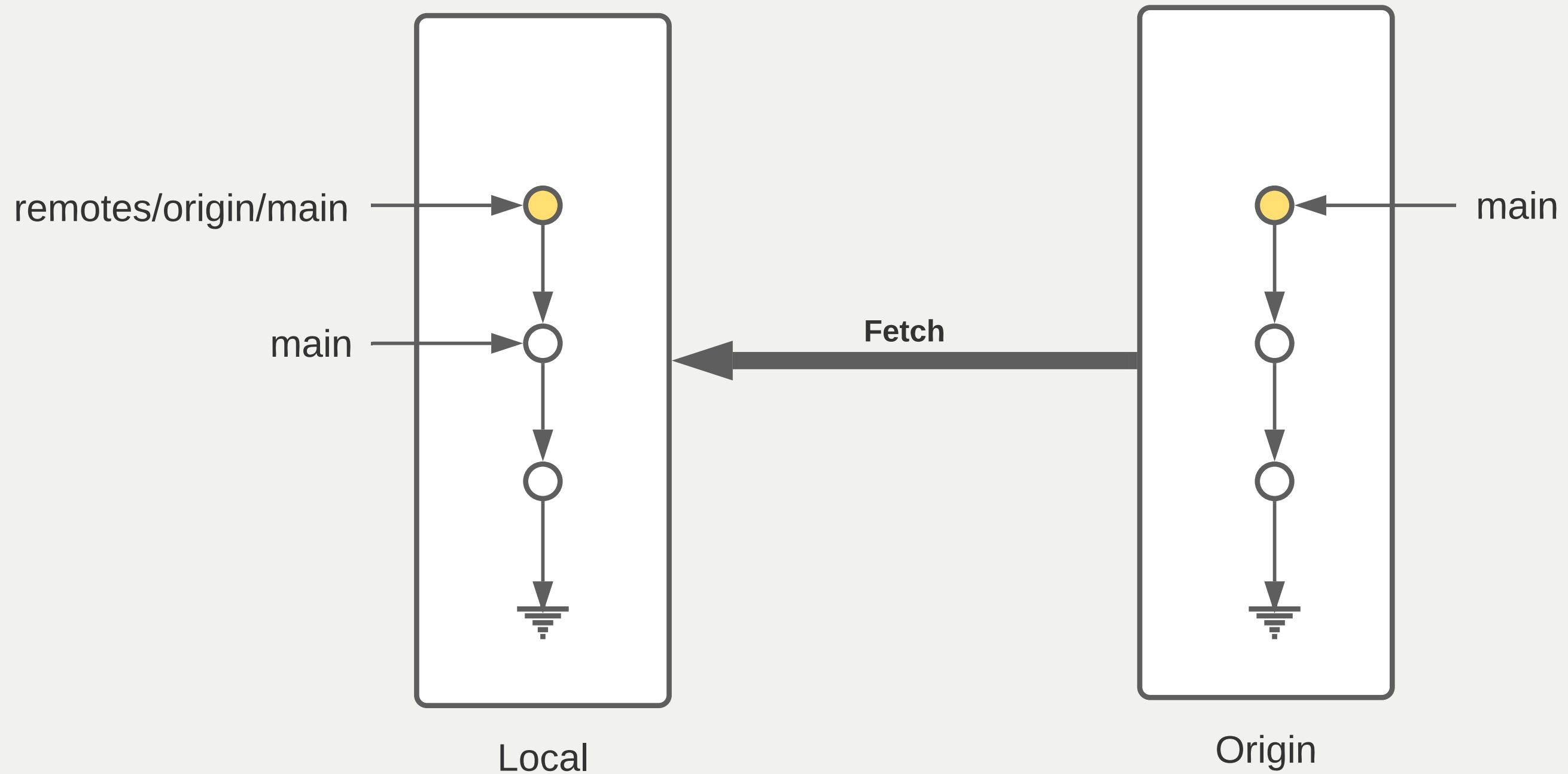
# Ouvrez votre README
code ./README.md

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

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

Copy





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



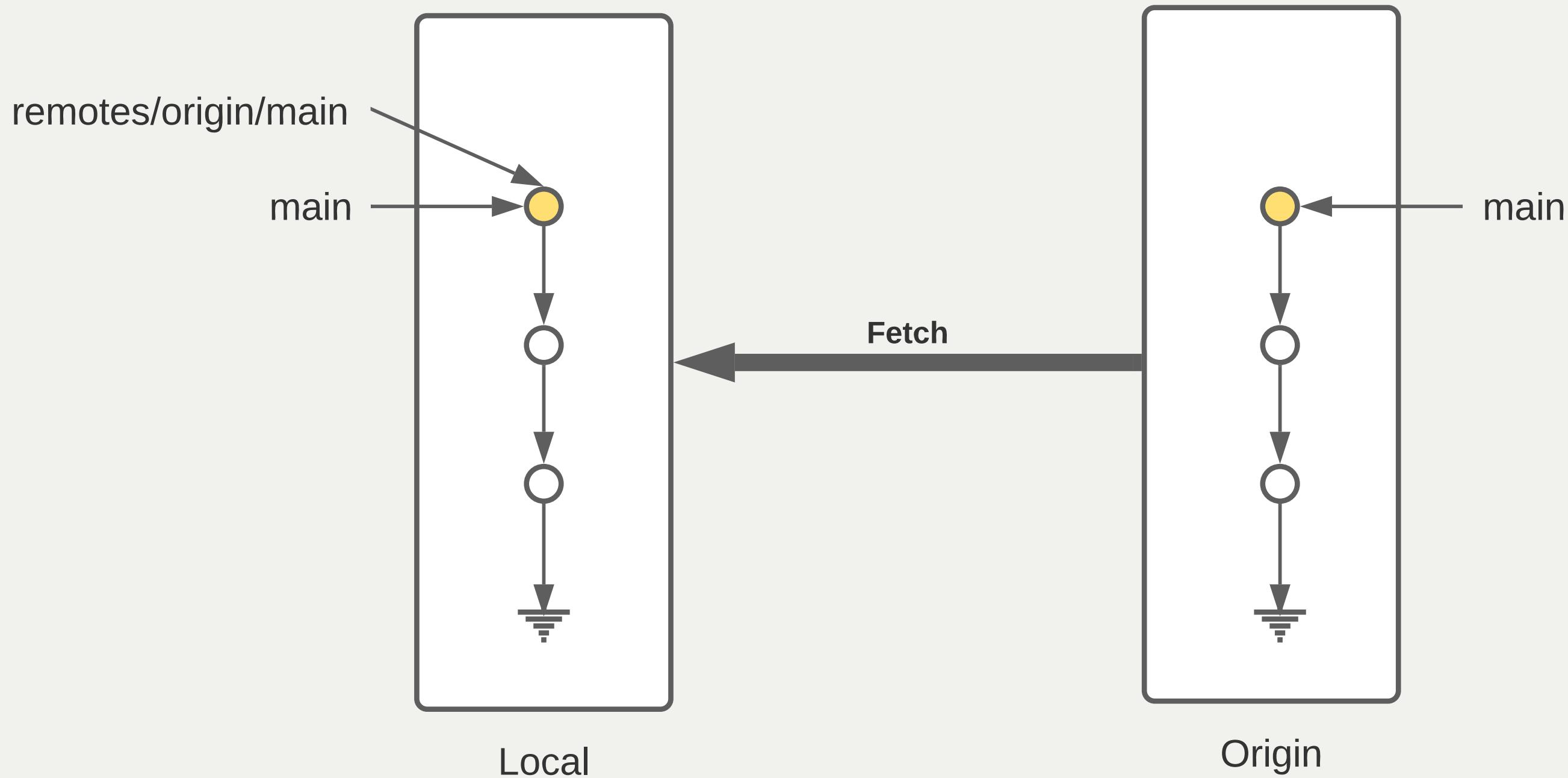
Vu que votre branche main n'a pas divergé (== partage le même historique) de la branche distante,
git merge effectue automatiquement un "fast forward".

```
Updating 1919673..b712a8e
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

Copy

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





```
# 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 fichier README.md



Git(Hub|Lab|teal...)

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|teal...) = 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)

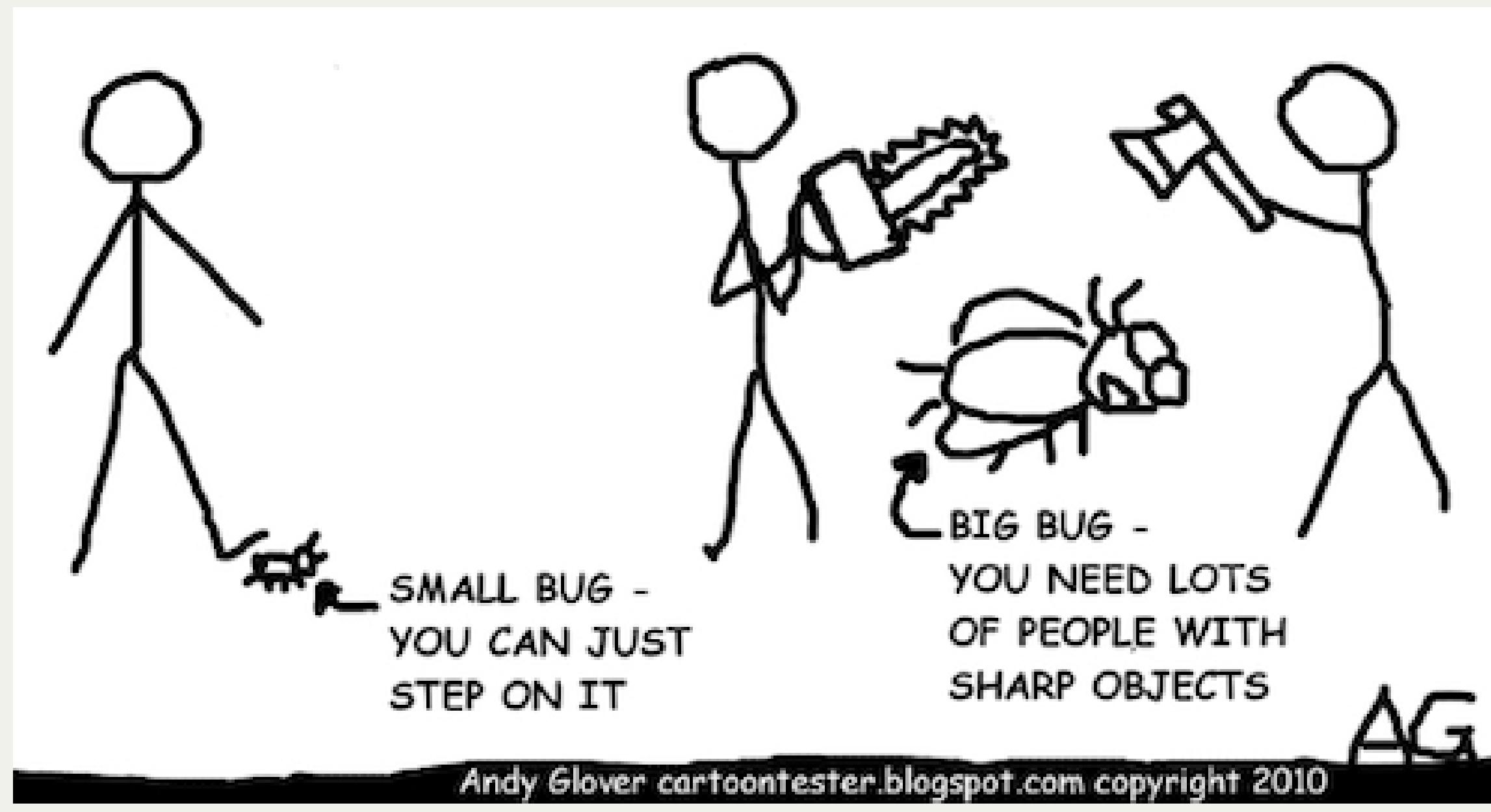
Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.

— Martin Fowler



Pourquoi la CI ?

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



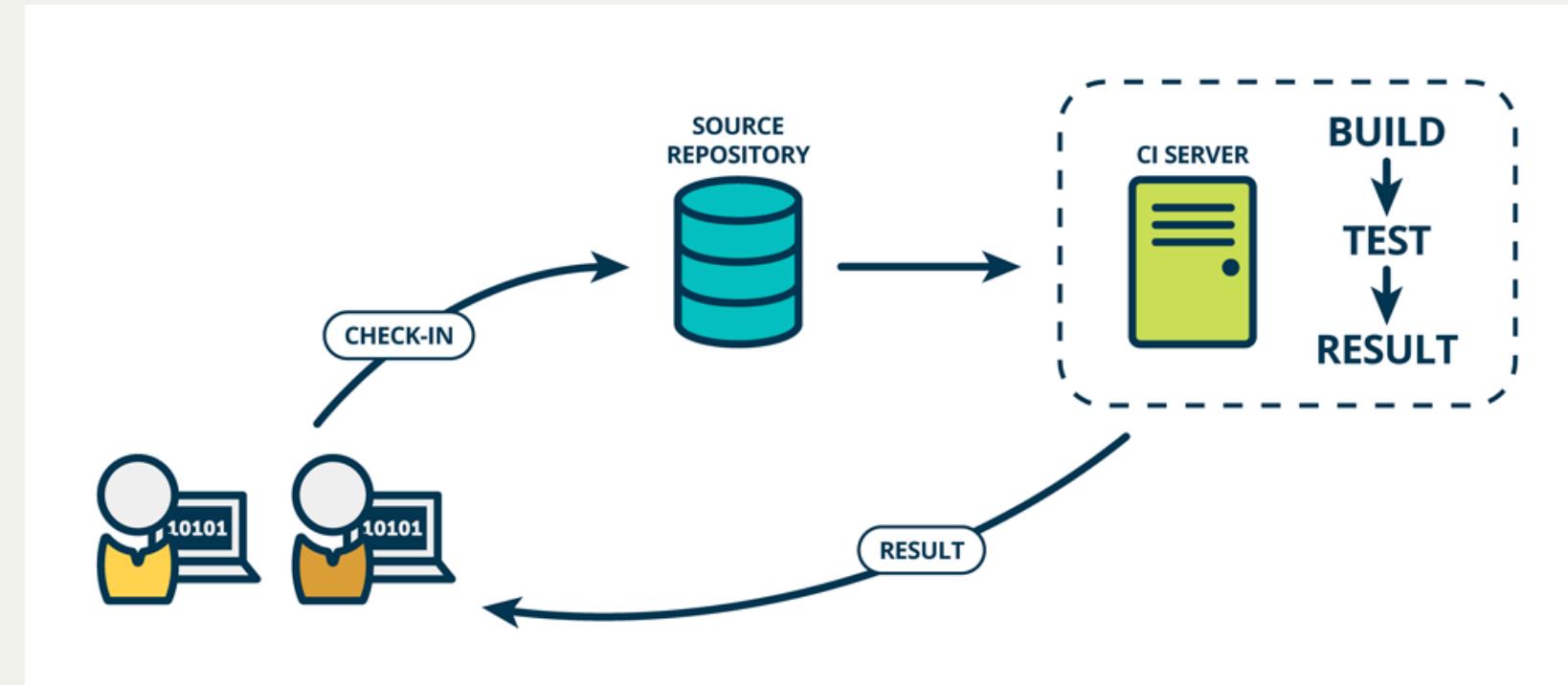
Qu'est ce que l'Intégration Continue ?

Objectif : que l'intégration de code soit un *non-événement*

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

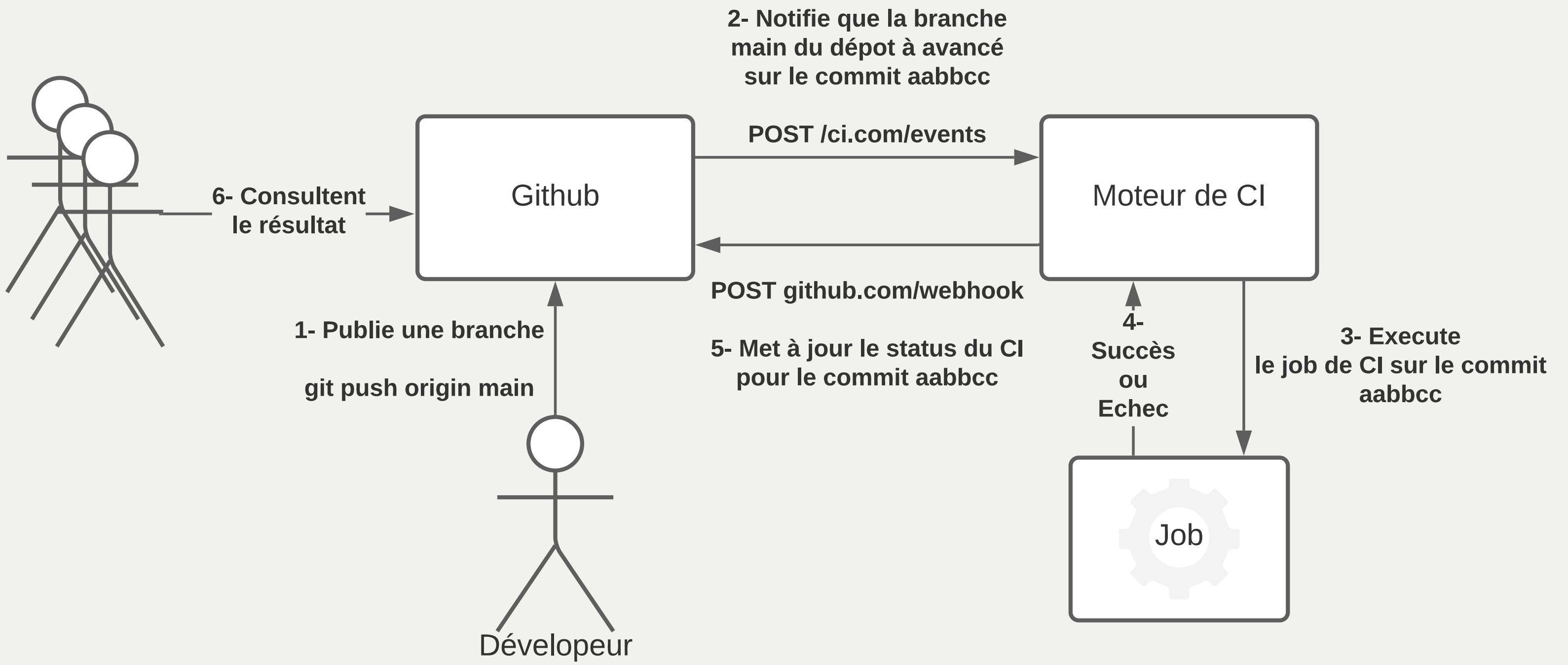


Et concrètement ? 1/2



- Un•e dévelopeu•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 dévelopeu•se•r•s

Et concrètement ? ↗



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



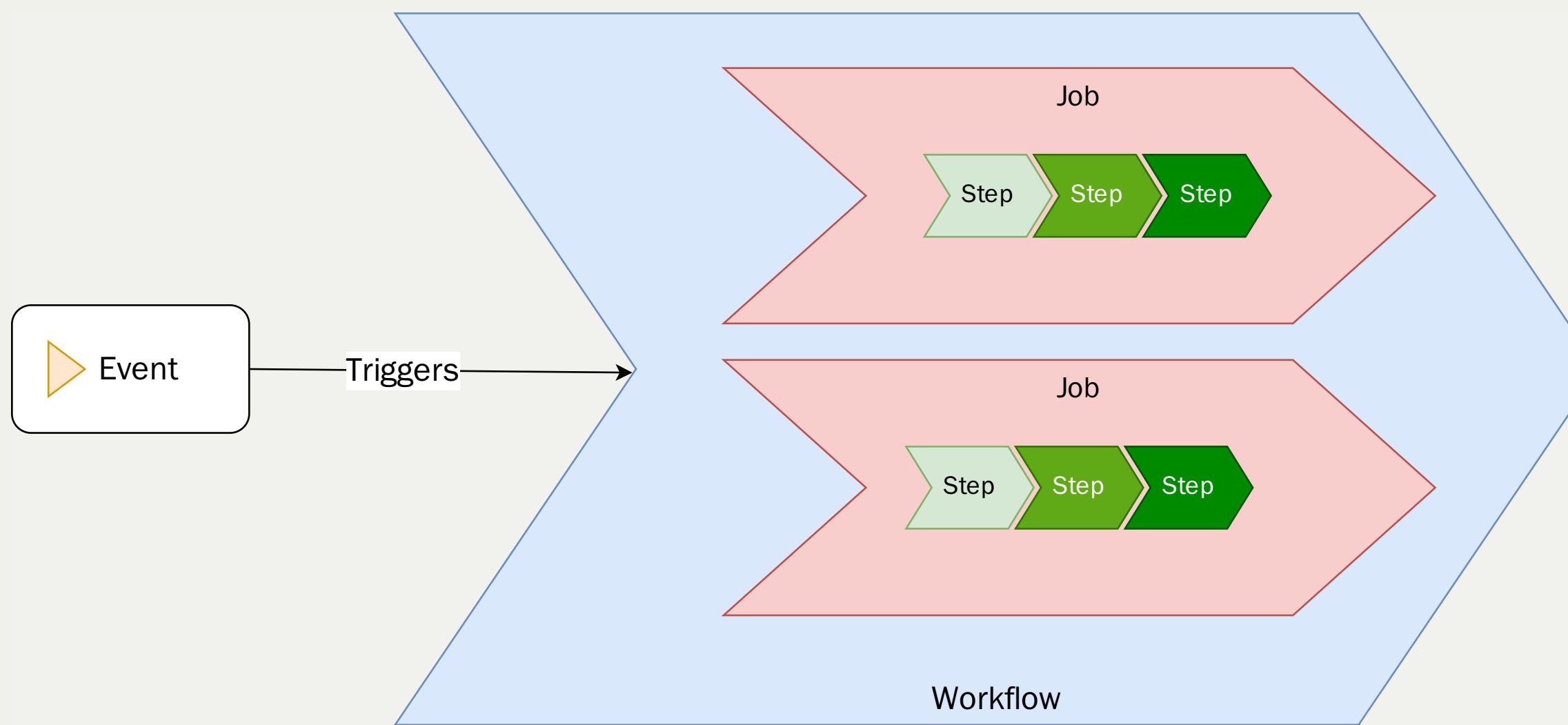
GitHub Actions

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

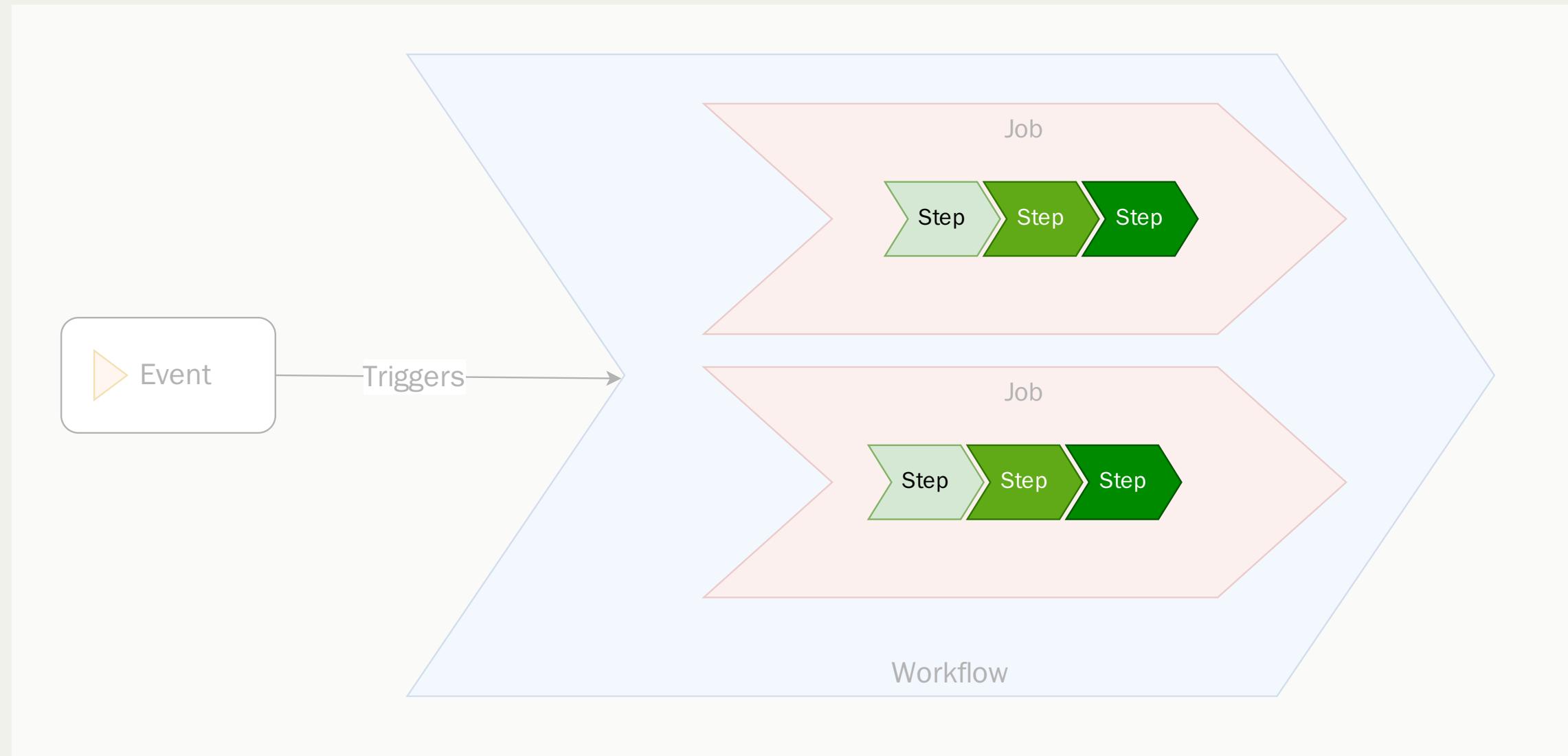
- ✓ : Très facile à mettre en place, gratuit et intégré complètement
- ✗ : Utilisable uniquement avec GitHub, et DANS la plateforme GitHub



Concepts de GitHub Actions



Concepts de GitHub Actions - Step 1/2



Concepts de GitHub Actions - Step 2/2

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

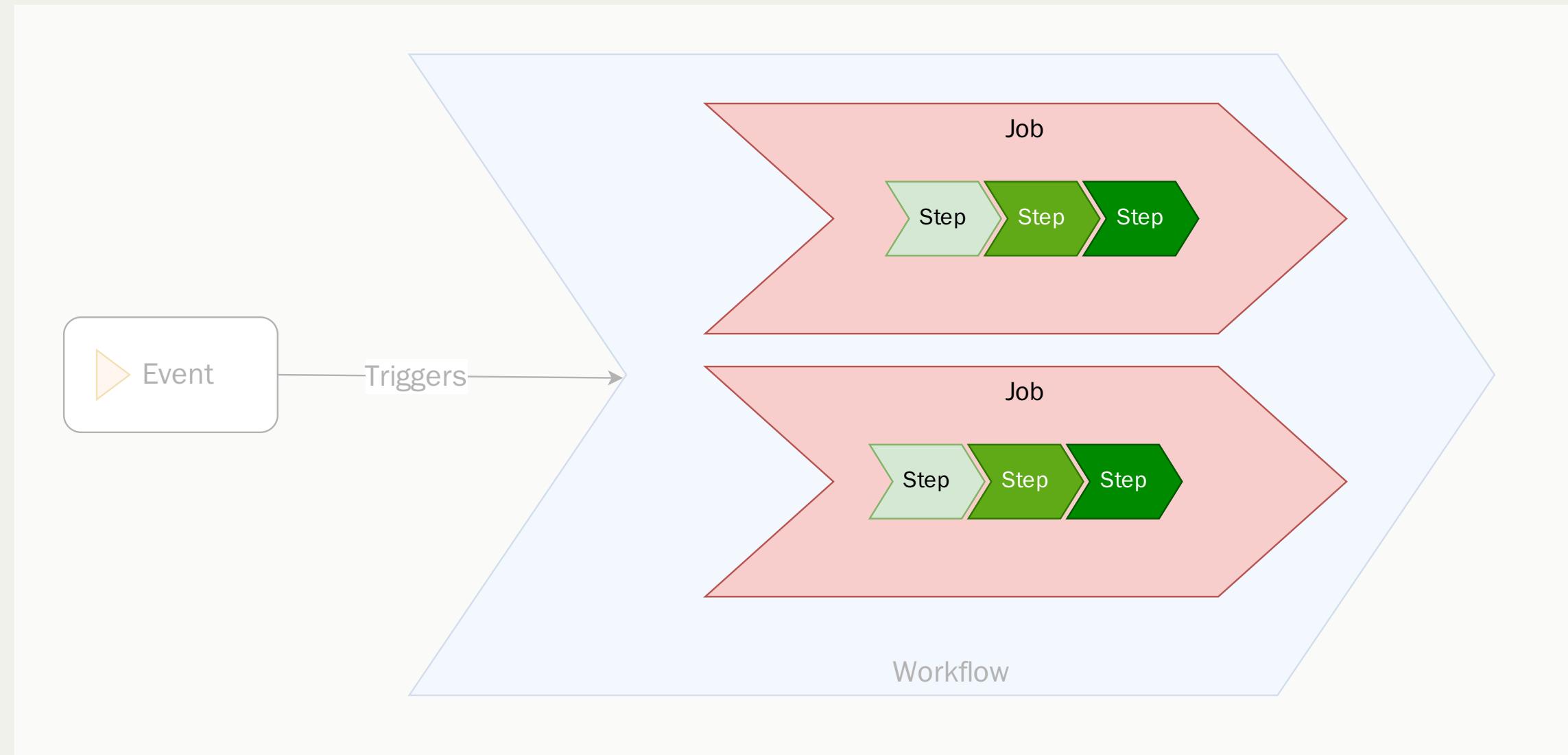
```
steps: # Liste de steps
  # Exemple de step 1 (commande)
  - name: Say Hello
    run: echo "Hello ENSG"
  # Exemple de step 2 (une action)
  - name: 'Login to DockerHub'
    uses: docker/login-action@v1 # https://github.com/marketplace/actions/docker-login
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME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

Copy

?

13.10

Concepts de GitHub Actions - Job 1/2



Concepts de GitHub Actions - Job 2/2

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

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

```
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

?

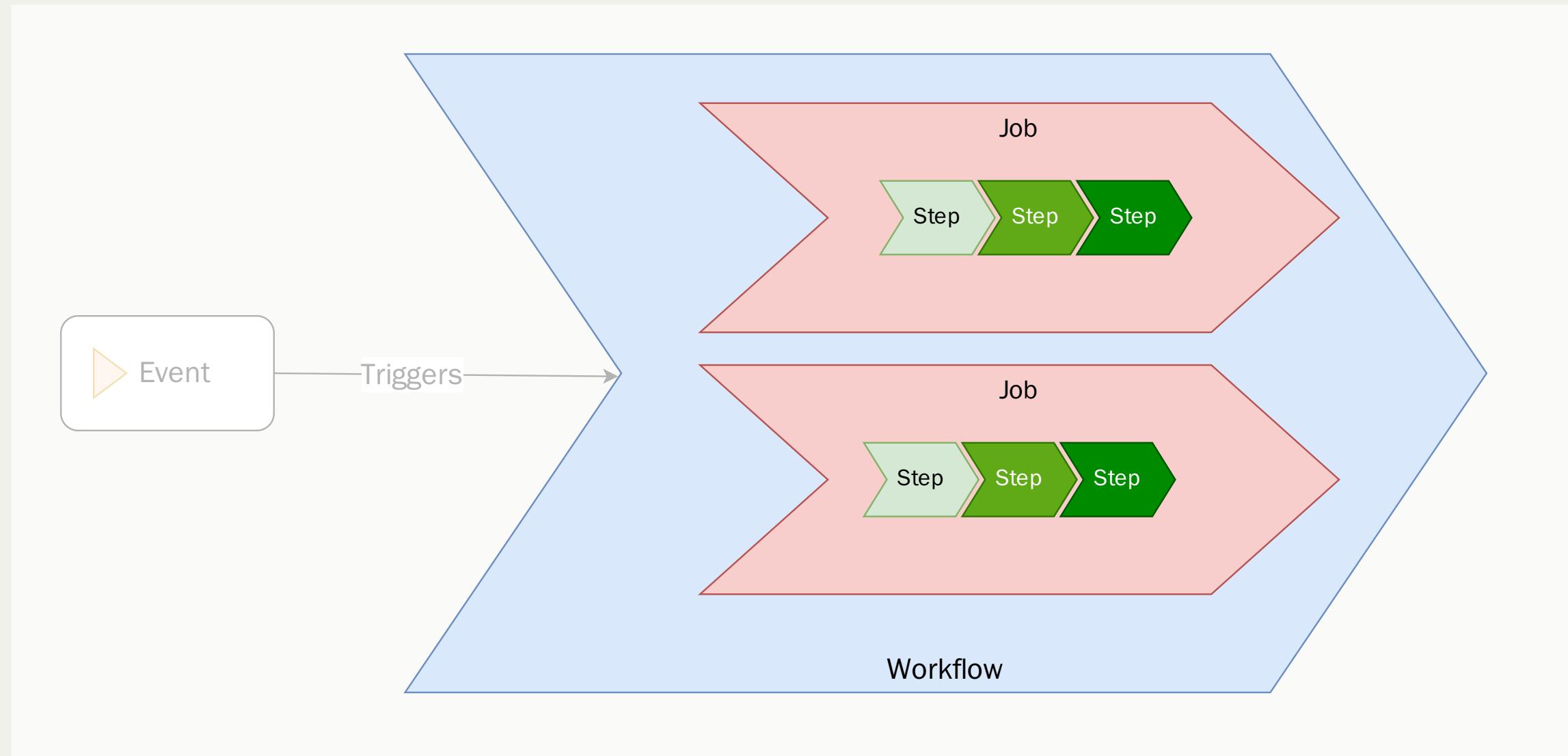
13.12

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.

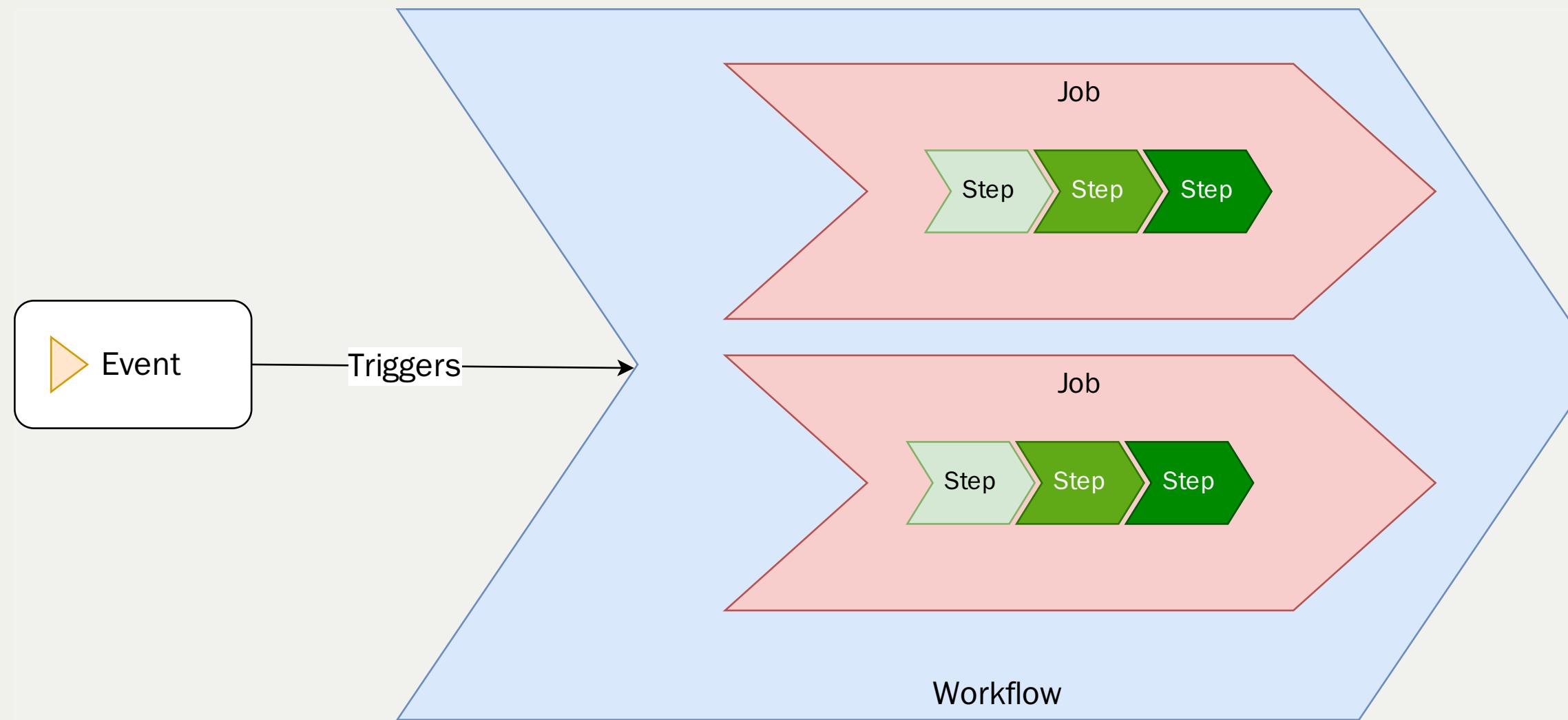
- On parle de "Workflow/Pipeline as Code"
- Chemin : .github/workflows/<nom du workflow>.yml
- On peut avoir *plusieurs* fichiers donc *plusieurs* workflows

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

Copy



Concepts de GitHub Actions - Évènement 1/2



Concepts de GitHub Actions - Évènement 2/2

Un **évenement** 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 :

```
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

?

13.18

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

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

Copy

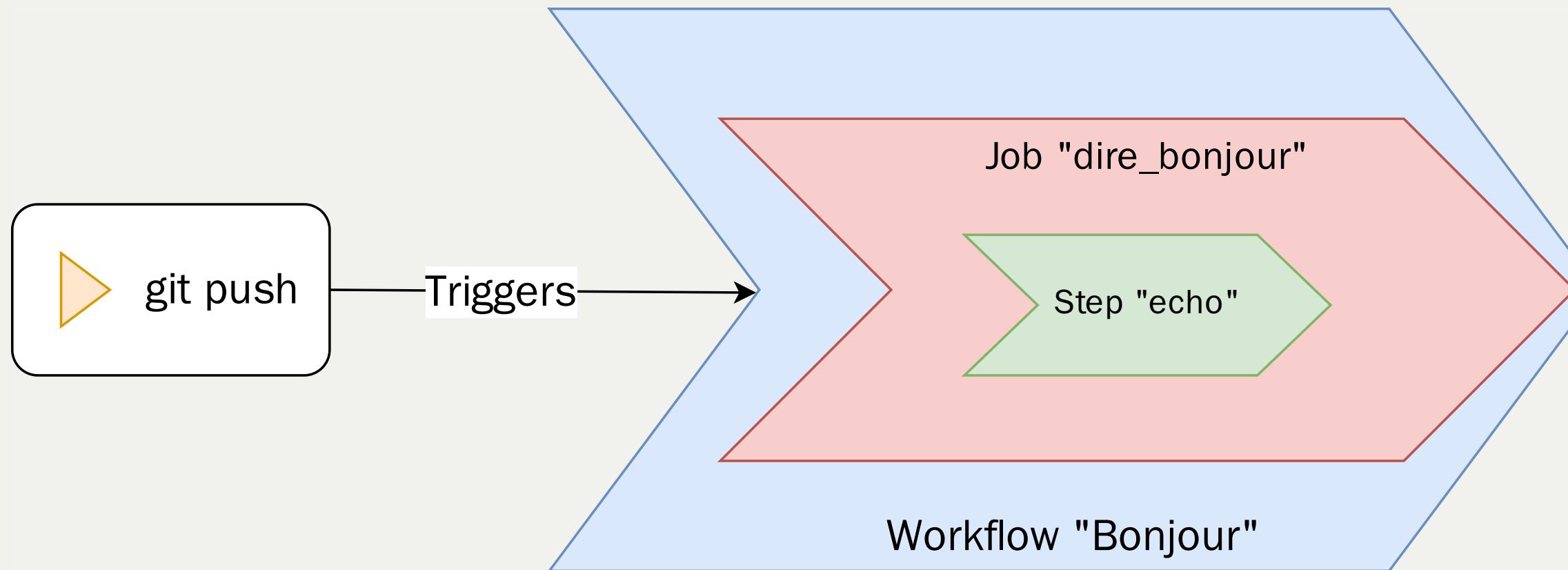
- Commitez puis poussez
- Revenez sur la page GitHub de votre projet et naviguez dans l'onglet "Actions" :



- Vouvez-vous un workflow ? Et un Job ? Et le message affiché par la commande echo ?

13 / 20

Exemple simple avec GitHub Actions : Récapète



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`:

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

Copy

- 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

```
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: (Δ sensible aux mises à jour, \checkmark facile à mettre en place)
 - Spécifier un environnement préfabriqué pour le workflow (Δ complexe, \checkmark portable)
 - Utiliser les fonctionnalités de votre outil de CI (Δ spécifique au moteur de CI, \checkmark 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

```
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
```

Copy





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

```
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
```

Copy



- 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

```
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 }}"
```

Copy



"menu-server"

 C'est à vous de modifier le projet "menu-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
- S'assurer d'utiliser les **même** versions de Java et Maven que dans Gitpod (💡 mvn -v)
 - 💡 <https://github.com/marketplace/actions/setup-java-jdk>
 - 💡 <https://github.com/marketplace/actions/setup-maven>
- L'application est compilée
- Le jar de l'application est fabriqué



"menu-server"

```
name: Menu Server CI
on:
  - push
jobs:
  menu_server:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4
      - name: Setup JDK Zulu 17
        uses: actions/setup-java@v4
        with:
          distribution: 'zulu'
          java-version: '17'
      - name: Setup Maven 3.9.0
        uses: stCarolas/setup-maven@v5
        with:
          maven-version: 3.9.0
      - name: Check Maven tooling
        run: mvn -v
      - name: Build application
        run: mvn compile
```

Copy

?

13 / 35



Checkpoint

- Pour chaque commit poussé dans la branche main de Menu 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 ?



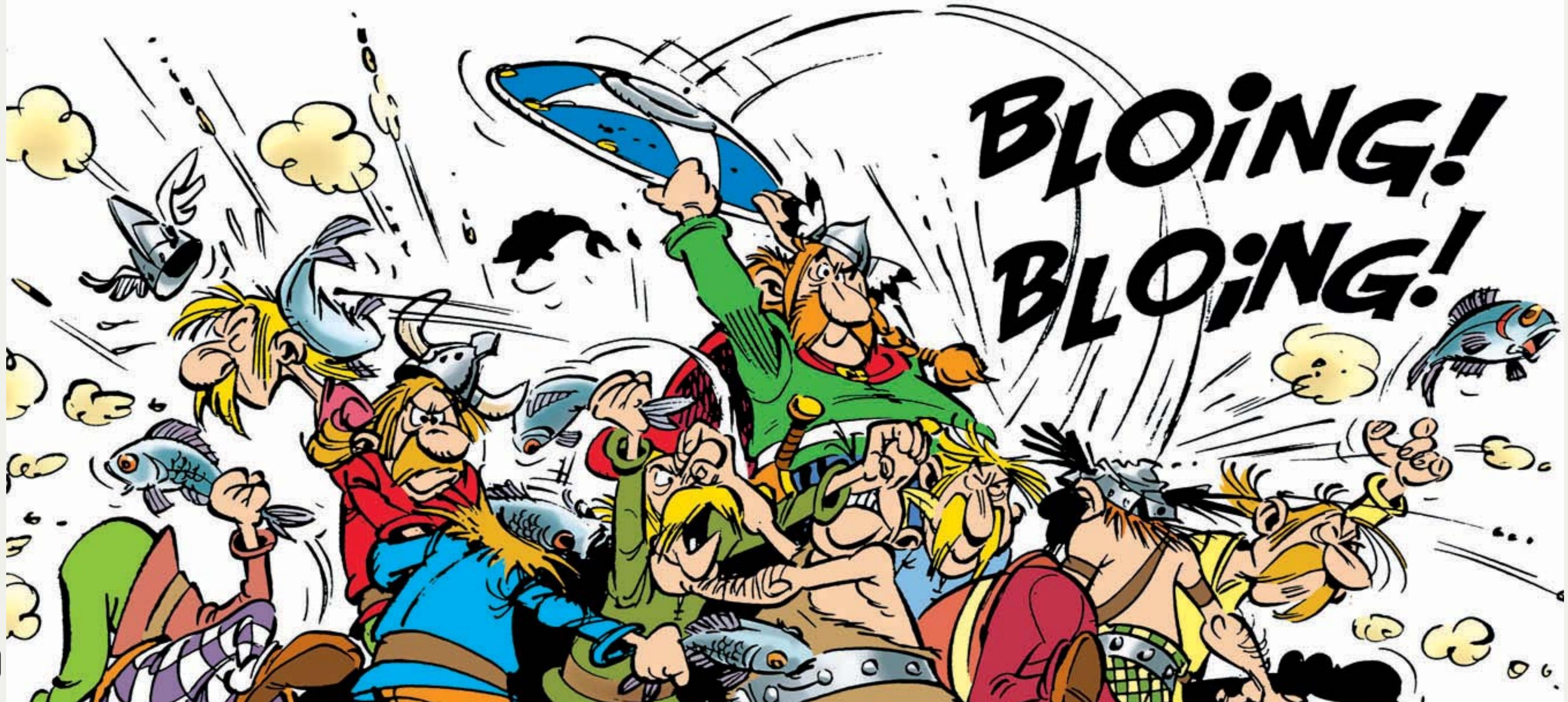
Collaborer c'est pas évident, mais il existe des outils et des méthodes pour vous aider.

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 ses camarades !
- C'est un outil très flexible... chacun peut faire ce qu'il lui semble bon !



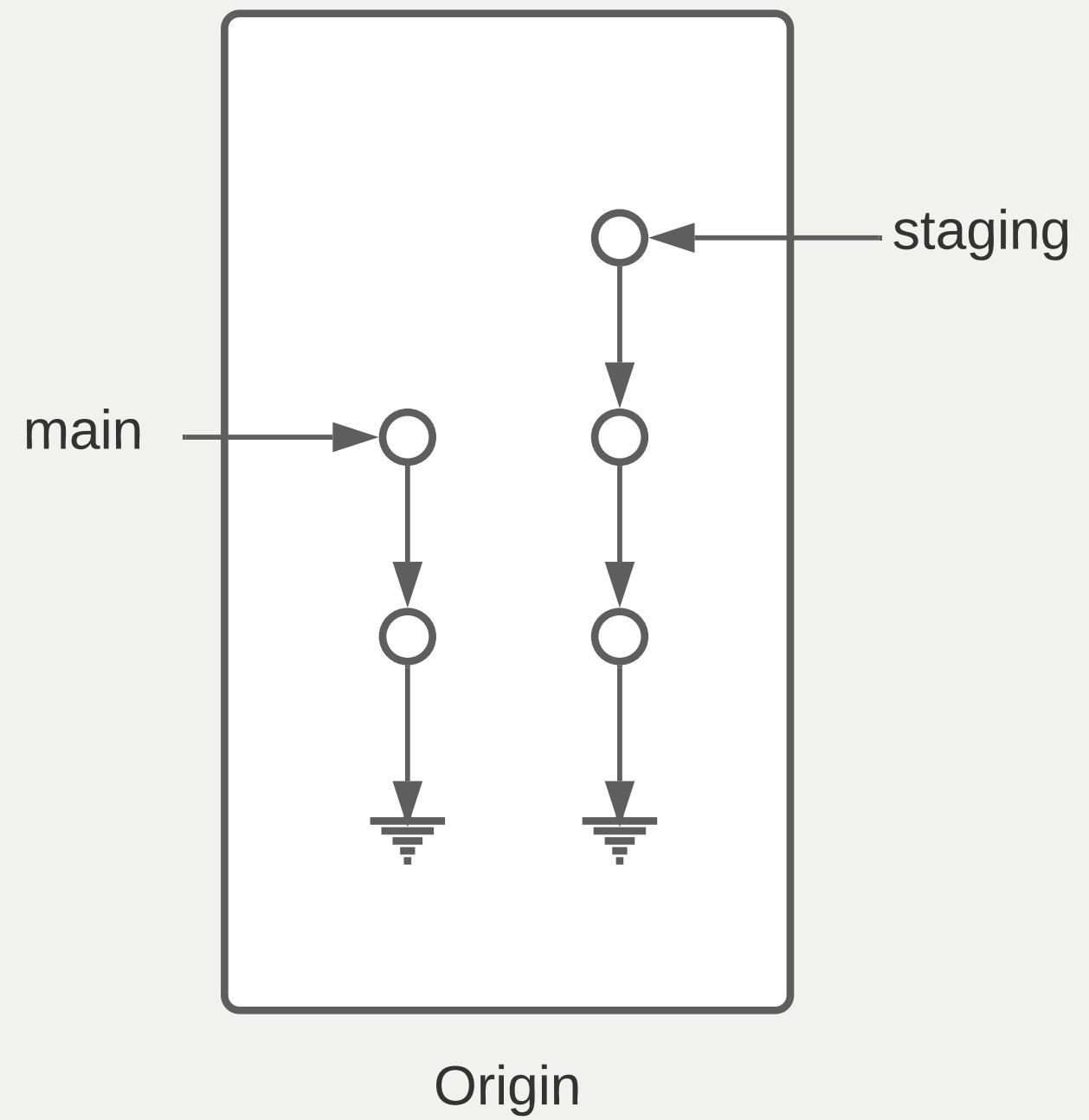
Un Example de Git Flow

(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

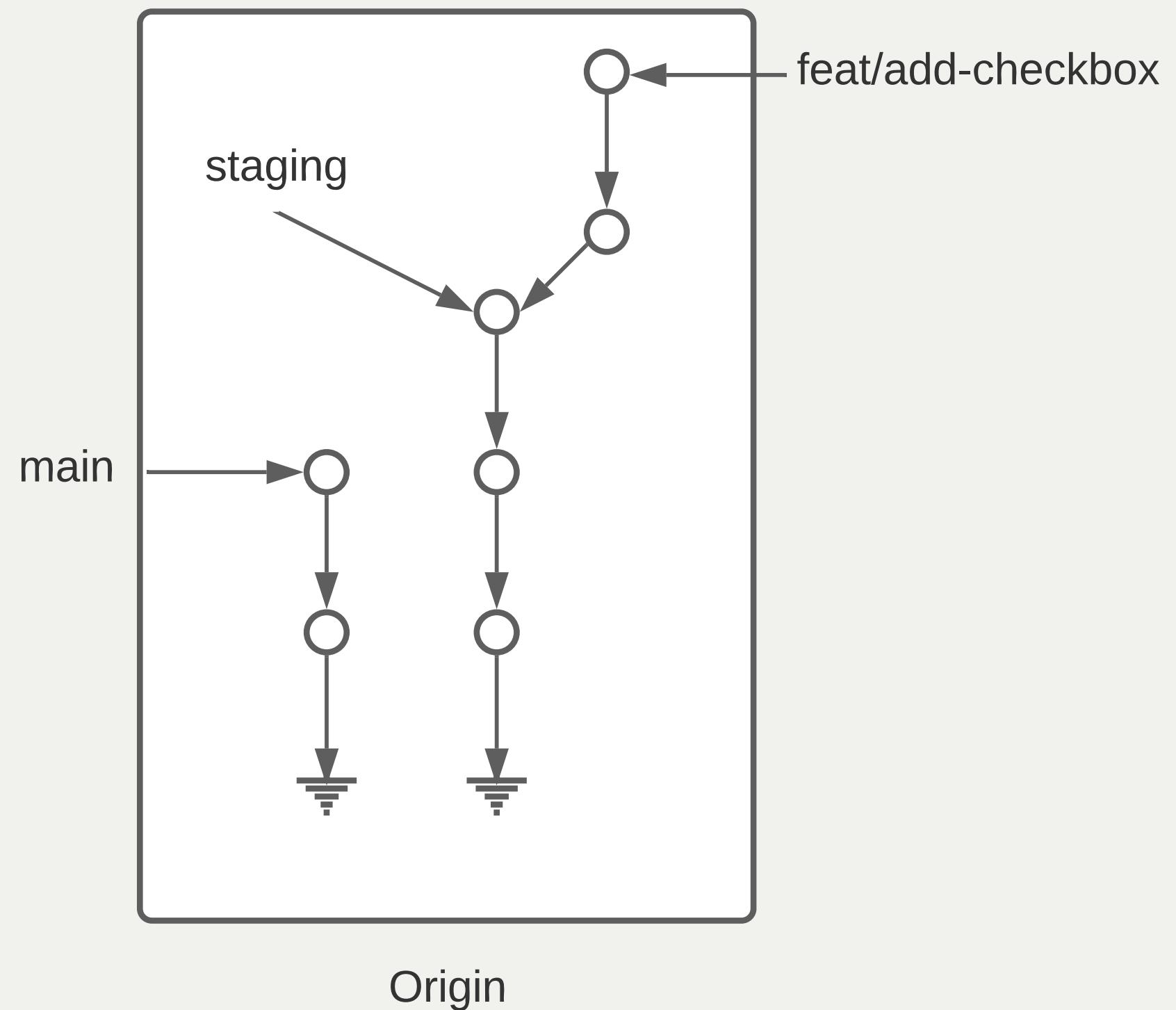


Origin



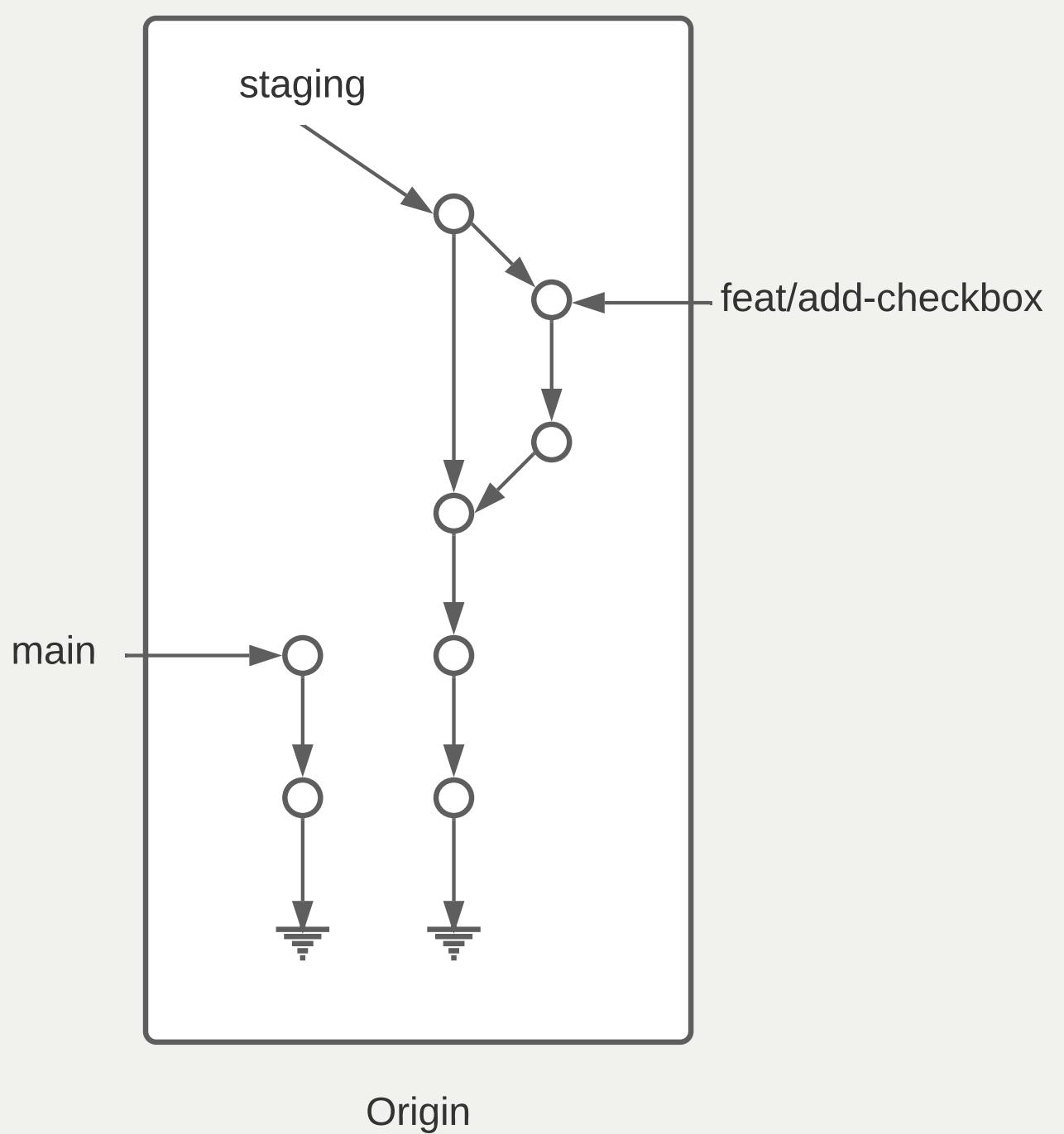
Gestion des branches

- 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

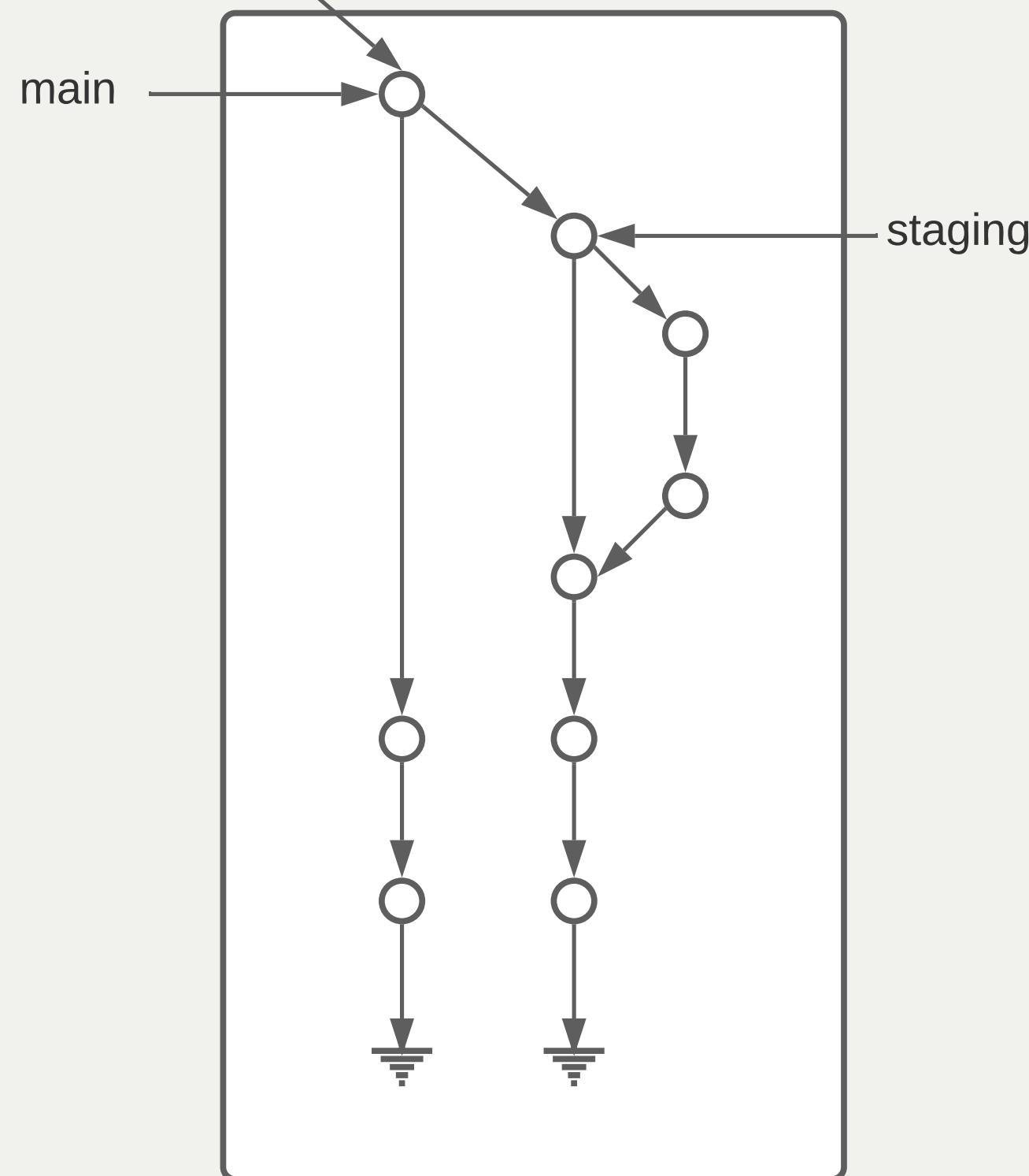




Origin

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



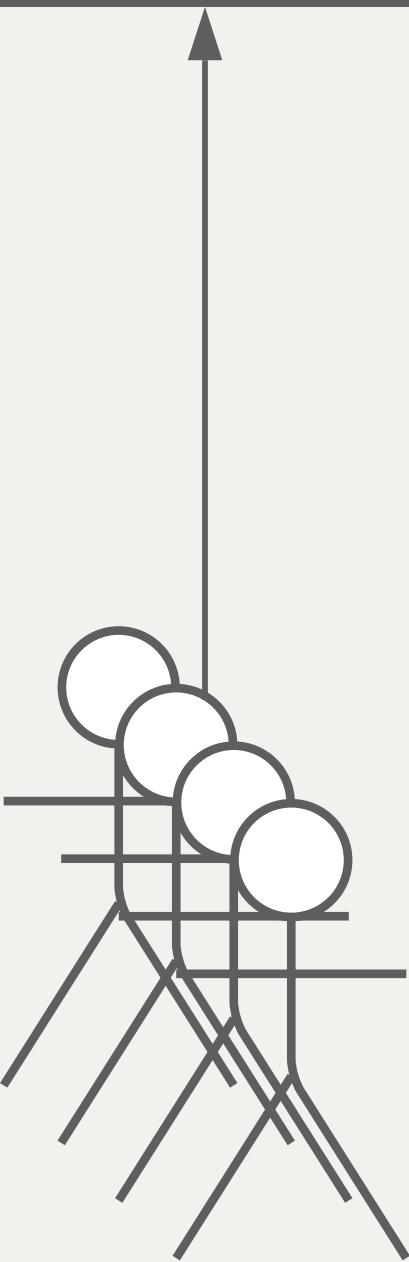
Un remote pour les gouverner tous

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

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

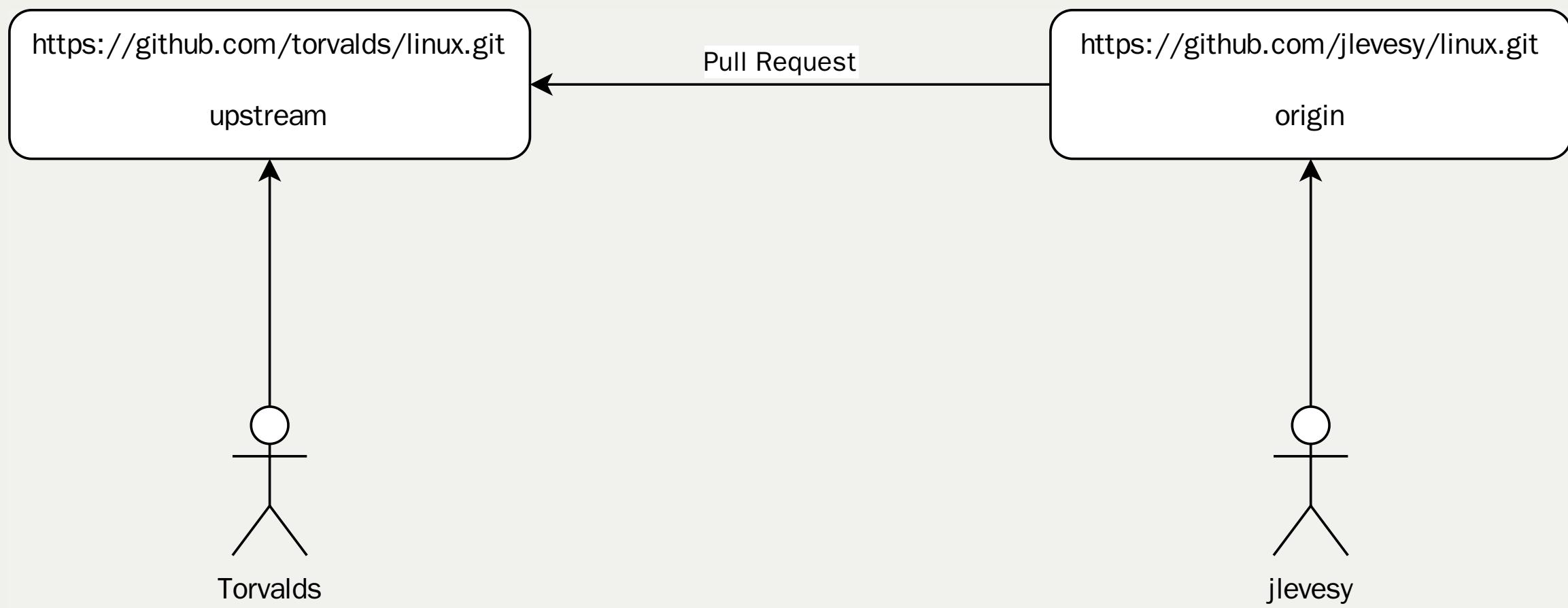
<https://github.com/torvalds/linux.git>

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 !

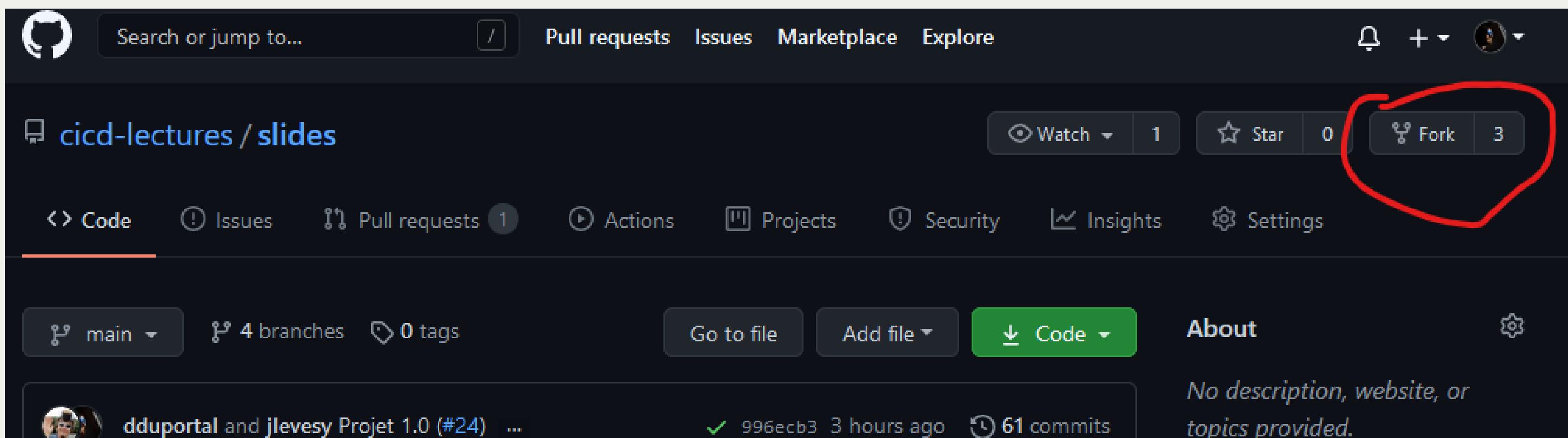


Forks ! Forks everywhere !

Dans la terminologie GitHub:

- Un fork est un remote copié d'un dépôt principal
 - C'est là 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: Ajoutez la fonctionnalité "suppression d'un menu" au projet de votre binôme





Exercice : Contribuez au projet de votre binôme (1/5)

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 implement-delete  
# Équivalent de git checkout -b <...>
```

Copy



binôme (2/5)

Maintenant voici la liste des choses à faire:

- Rajouter le MenuRepository comme dépendance du MenuController
- Implémenter une nouvelle méthode deleteMenu
 - Gère les requêtes DELETE /menus/{id}
 - Appelle la méthode deleteById du menuRepository
 - Réponds 200 si la suppression est réussie
- Bonus si vous arrivez à faire en sorte que le serveur réponde 404 si un menu à supprimer n'existe





Exercice : Contribuez au projet de votre binôme (3/5)

Pour tester votre changement

```
# D'abord on crée un menu
```

curl -H "Content-Type: application/json" --data-raw '{"name": "Menu spécial du chef", "dishes": [{"name": "Bananes aux fr

Copy

```
# Puis on le supprime
```

curl -XDELETE localhost:8080/menus/4

Copy

```
# Et on vérifie que le menu est bien supprimé
```

curl localhost:8080/menus

Copy



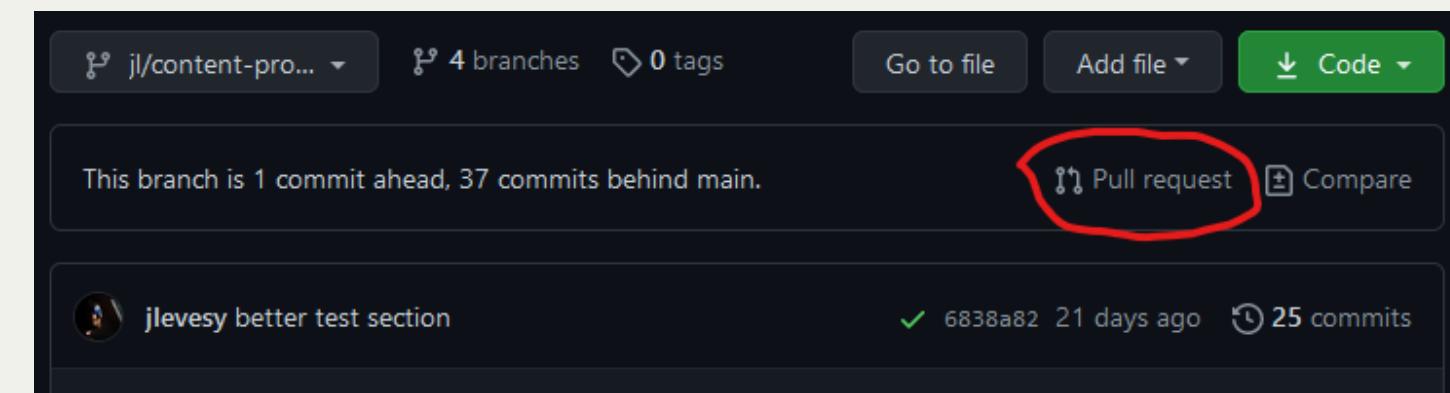
Exercice : Contribuez au projet de votre binôme (4/5)

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 (5/5)

Dernière étape: ouvrir une pull request!

- Rendez vous sur la page de votre projet
- Sélectionnez votre branche dans le menu déroulant "branches" en haut à 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 être 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 spec de votre workflow pour qu'il se déclenche aussi sur une PR
- Vous pouvez changer la spec du workflow directement dans votre PR
- La documentation se trouve par ici



Checkpoint

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

Même si le linter "ilécon", même si on a la flemme et "sépanou" qui avons cassé le CI.

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 a tout moment.
- Déetecte les impacts non anticipés des changements introduits
- Evite l'introduction de régressions
- Écrire des tests est un acte préventif et non curatif.



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.
- Par exemple :
 - Retourne les bonnes valeur en fonction des paramètres donnés
 - Appelle la bonne méthode du bon attribut avec les bons paramètres



Mise en place de l'exercice

- Depuis votre environnement de développement, dans le répertoire du **fork** de votre binôme
- Créez une feature branch add-tests.



TODO REFAIRE TOUT CA avec go.



Ajout des Outils de Tests Automatisés au Projet (3/3)

- Exécutez les tests unitaires avec la commande mvn test
 - Spoiler : No tests to run...
 - Pourquoi ca ?





Exercice : Corriger un Bug (1/11)

- La classe `ListMenuService` semble être "buggée" ...
 - Tous les noms des menus sont **TEST TODO** 😱
- Quand on regarde l'implémentation, on se rends compte que le problème provient de la méthode statique `fromModel` de la classe `MenuDto`
- Même si la correction est aisée, on va d'abord écrire un test unitaire qui valide le comportement du service.
- Notre SUT: `ListMenuService` + `DTO` + `Model`



Exercice : Corriger un Bug (2/11)

Mise en place du test

```
// src/test/java/com/cicdlectures/menuserver/service/ListMenuServiceTests.java
```

```
public class ListMenuServiceTests {  
  
    private ListMenuService subject;  
  
    @BeforeEach  
    public void init() {  
        subject = new ListMenuService(null);  
    }  
  
    @Test  
    @DisplayName("lists all known menus")  
    public void listsKnownMenus() {  
        List<MenuDto> got = subject.listMenus();  
    }  
}
```

Copy





Exercice : Corriger un Bug (3/11)

- Super on à un test, il ne reste plus qu'à le lancer avec mvn test 🎉
- Spoiler java.lang.NullPointerException





Exercice : Corriger un Bug (4/11)

- Le ListMenuService à besoin d'un MenuRepository pour fonctionner.
- Cependant :
 - On ne veut pas valider le comportement du MenuRepository, il est en dehors de notre SUT.
 - Pire, on ne veut pas se connecter à une base de donnée pendant un test unitaire.



Exercice : Corriger un Bug (5/11)

Solution : On fournit une "fausse implémentation" au service, un mock.

```
// src/test/java/com/cicdlectures/menuserver/service/ListMenuServiceTests.java
```

private MenuRepository menuRepository;

private ListMenuService subject;

@BeforeEach

```
public void init() {  
    this.menuRepository = mock(MenuRepository.class);  
    this.subject = new ListMenuService(this.menuRepository);  
}
```

Copy



Exercice : Corriger un Bug (6/11)

Ce "mock" peut être piloté dans les tests!

```
@Test  
 @DisplayName("lists all known menus")  
 public void listsKnownMenus() {  
     // Quand le repository reçoit l'appel findAll  
     // Alors il retourne la valeur null.  
     when(menuRepository.findAll()).thenReturn(null);  
 }
```

Copy



Exercice : Corriger un Bug (7/11)

- Super on a un test unitaire, il ne reste plus qu'à le lancer avec mvn test 🎉
- Spoiler: ✓



Sauf qu'on avait pas un bug à corriger au fait?





Exercice : Corriger un Bug (8/11)

Objectif: Vérifier que les valeurs retournées par le ListMenuService sont cohérentes avec les données en base, pour cela il nous faut:

- Préparer un jeu de données de test et configurer le mock du repository pour qu'il le retourne
- Appeler notre service
- Comparer le résultat obtenu du service avec des valeurs attendues.

```
@Test
@DisplayName("lists all known menus")
public void listsKnownMenus() {
    // Défini une liste de menus avec un menus.
    Iterable<Menu> existingMenus = Arrays.asList(
        new Menu(
            Long.valueOf(1),
            "Christmas menu",
            new HashSet<>(
                Arrays.asList(
                    new Dish(Long.valueOf(1), "Turkey", null),
                    new Dish(Long.valueOf(2), "Pecan Pie", null)
                )
            )
        )
    );
}

// On configure le menuRepository pour qu'il retourne notre liste de menus.
when(menuRepository.findAll()).thenReturn(existingMenus);

// On appelle notre sujet
List<MenuDto> gotMenus = subject.listMenus();

// On défini wantMenus, les résultats attendus
Iterable<MenuDto> wantMenus = Arrays.asList(
    new MenuDto(
        Long.valueOf(1),
```

Exercice : corriger un bug (10/11)

- Super on a un test unitaire (qui teste!), il ne reste plus qu'à le lancer avec mvn test 🎉
- Spoiler:

```
[ERROR] Failures:  
[ERROR]   ListMenuServiceTests.listsKnownMenus:66  
expected:  
  <[MenuDto(id=1, name=Christmas menu, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey) ]) ]>  
but was:  
  <[MenuDto(id=1, name=TEST TODO, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey) ]) ]>
```

Copy



15 . 20

- Il ne reste plus qu'à faire la correction et le tour est joué!



Test Unitaire : Quelques Règles

- Un test unitaire teste un et un seul comportement
- Faites attention a ce que votre test teste vraiment quelque chose!
 - Avec les mocks, c'est facile de se faire piéger.
- Essayez, dans la mesure du possible, d'écrire vos tests (qui échouent) avant d'écrire votre code.
- Il n'y a pas de définition ferme du SUT
 - Attention à garder une taille raisonnable (quelques classes).
 - Privilégiez les tests de méthodes publiques.



Checkpoint

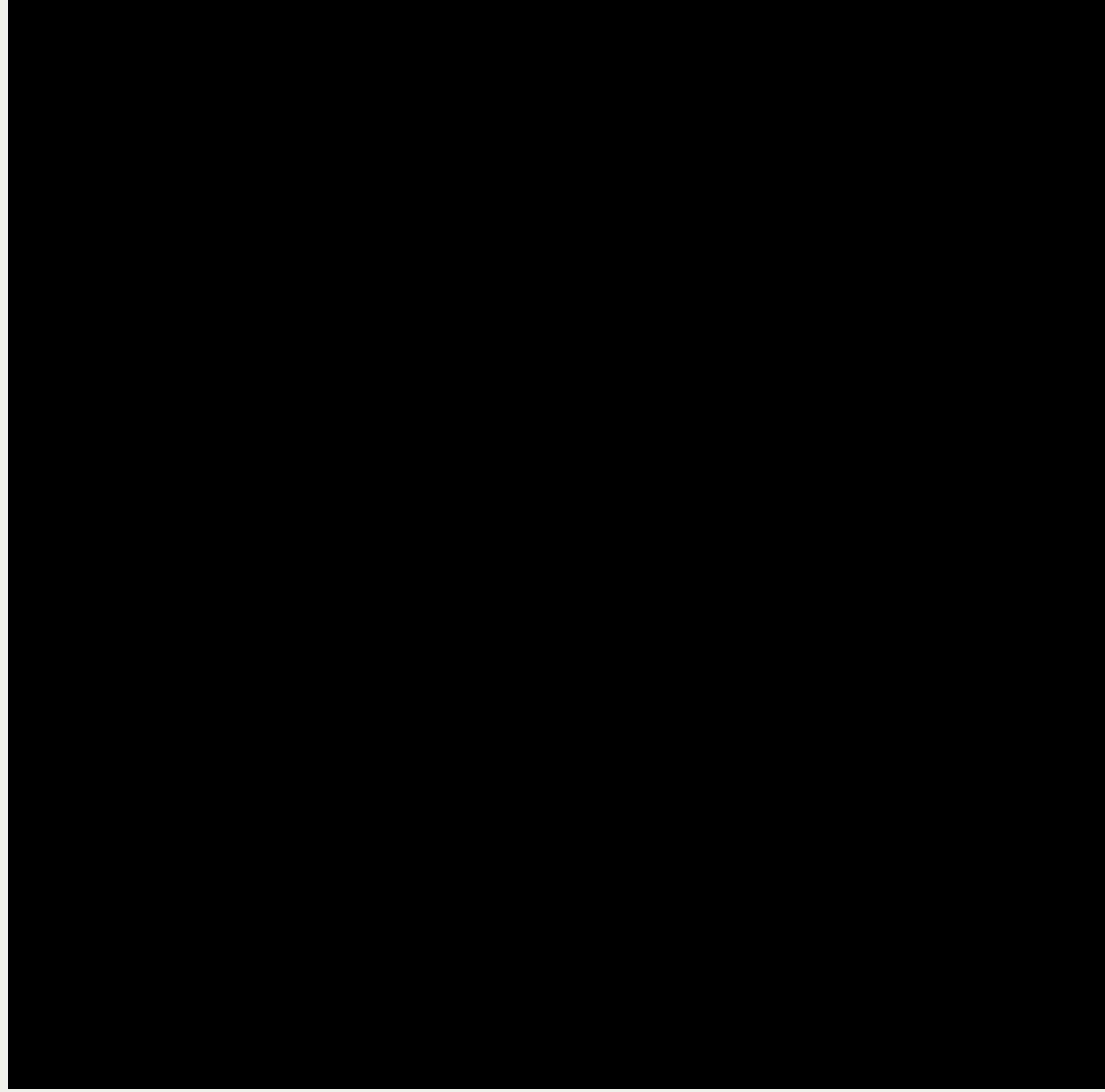


On a vu :

- 🔎 Qu'il faut tester son code
- 🌎 Qu'il existe différents type de tests en fonction de ce que l'on veut tester
- 🧩 Comment faire des tests unitaires

Test Unitaire : Pro / Cons

- ✓ Super rapides (<1s) et légers à exécuter
- ✓ Pousse à avoir un bon design de code
- ✓ Efficaces pour tester des cas limites
- ✗ Peu réalistes



?, ☰, ⌂

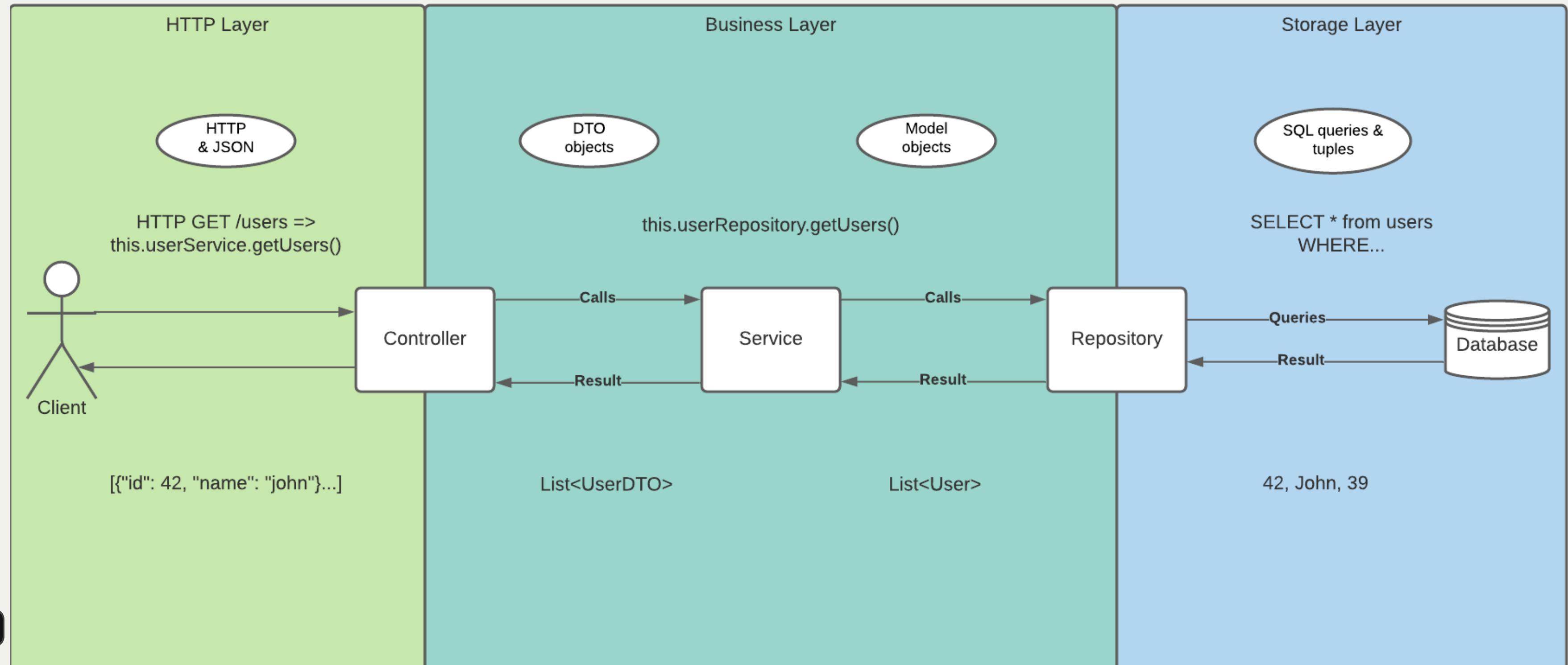
Tester des composants indépendamment ne prouve pas que le système fonctionne une fois intégré!



✓ Solution: Tests d'intégration

- Test validant qu'un assemblage d'unités se comportent comme prévu.
- Teste votre application au travers de toutes ses couches
- Par exemple avec menu server:
 - Prouve que GET /menus retourne la liste des menus enregistrés en base
 - Prouve que POST /menus enregistre un nouveau menu en base avec ses plats.

Définition du SUI (1/2)



Définition du SUT (2/2)

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



Ce sont des tests plus lents et plus complexes que des tests unitaires. Comment gérer ça?



Exécuter Les Tests d'Intégration: Cycle de Vie Maven

- Les tests d'intégration sont une autre partie du cycle de vie de l'application: la phase `verify`.
- `verify` est une méta-phase composée de 3 sous-phases :
 - `pre-integration-test`: prépare l'environnement des tests d'intégration
 - `integration-test`: execute la suite de tests d'intégration
 - `post-integration-test`: nettoie l'environnement des tests d'intégration

⚠ Il faut toujours appeler `verify` et non pas `integration-test`, sinon la sous-phase `post-integration-test` ne s'exécutera pas ⚠



Exécuter Les Tests d'Intégration: Le Plugin failsafe (1/3)

- Pour exécuter les tests d'intégration nous allons introduire un nouveau plugin: failsafe
- Ce plugin exécute les tests ayant le suffixe `IT.java` (par exemple: `MaClasseIT.java`)
- Ce plugin s'exécute lors de la phase `integration-test`

Exécuter Les Tests d'Intégration: Le Plugin failsafe (2/3)

- Configurez le plugin Maven Spring Boot pour les tests d'intégration (section <build> <plugins>):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <configuration>
    <skipTests>${skipIntegrationTests}</skipTests>
  </configuration>
</plugin>
```

Copy



Exécuter Les Tests d'Intégration: Le Plugin failsafe (2/3)

Cela crée les commandes suivantes:

- mvn test: lance les tests unitaires
- mvn verify: lance les tests unitaires et d'intégration
- mvn verify -DskipUnitTests=true: lance uniquement les tests d'intégration

Tests d'Intégrations: Et concrètement avec le menu-server?

- Dans les faits... nous n'allons pas utiliser les phases pre-integration-test et post-integration-test
 - → Nous n'avons pas de serveur de base de données à démarrer.
 - → SpringBoot intègre le démarrage et l'arrêt du serveur web dans l'exécution des tests via l'annotation `@SpringBootTest`.
- C'est un projet pédagogique!
 - Dans un "vrai" projet, on voudrait peut-être démarrer / éteindre un serveur de base de données dans ces étapes.

Nous allons écrire un test d'intégration pour l'appel GET /menus





Exercice : Ecrire un test d'intégration (1/4)

Mise en place d'un test vide

```
// src/test/java/com/cicdlectures/menuserver/controller/MenuControllerIT.java
// Lance l'application sur un port aléatoire.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
// Indique de relancer l'application à chaque test.
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
public class MenuControllerIT {

    @LocalServerPort
    private int port;

    private URL getMenusURL() throws Exception {
        return new URL("http://localhost:" + port + "/menus");
    }

    @Test
    @DisplayName("lists all known menus")
    public void listsAllMenus() throws Exception {
    }
}
```

[Copy](#)



Exercice : Ecrire un test d'intégration (2/4)

Maintenant, on appelle le serveur et on vérifie que l'appelle qu'il nous répond une 200

```
// src/test/java/com/cicdlectures/menusever/controller/MenuControllerIT.java
// Lance l'application sur un port aléatoire.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
// Indique de relancer l'application à chaque test.
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
public class MenuControllerIT {
    // ...

    private RestTemplate template = new RestTemplate();

    @Test
    @DisplayName("lists all known menus")
    public void listsAllMenus() throws Exception {
        ResponseEntity<MenuDto[]> response = this.template.getForEntity(getMenusURL().toString(), MenuDto[].class);

        assertEquals(HttpStatus.OK, response.getStatusCode());
    }
}
```

Copy



?



Copy

Bon, c'est bien sympa mais notre test n'est pas satisfaisant en l'état. Il faut maintenant valider notre comportement principal: lister tous les menus connus





Exercice : Ecrire un test d'intégration (3/4)

D'abord il faut provisionner des données en base.

```
public class MenuControllerIT {  
    // ...  
    // Injecte automatiquement l'instance du menu repository  
    @Autowired  
    private MenuRepository menuRepository;  
  
    private final List<Menu> existingMenus = Arrays.asList(  
        new Menu(null, "Christmas menu", new HashSet<>(Arrays.asList(new Dish(null, "Turkey", null), new Dish(null, "Pecan  
        new Menu(null, "New year's eve menu", new HashSet<>(Arrays.asList(new Dish(null, "Potatos", null), new Dish(null, "  
  
    @BeforeEach  
    public void initDataset() {  
        for (Menu menu : existingMenus) {  
            menuRepository.save(menu);  
        }  
    }  
  
    // ...  
}
```

Copy

?

Il ne nous reste qu'a changer le corps du test pour verifier que le contenu de la reponse est celui auquel on s'attends.

```
public class MenuControllerIT {  
    // ...  
  
    @Test  
    @DisplayName("lists all known menus")  
    public void listsAllMenus() throws Exception {  
        // On declare la valeur attendue.  
        MenuDto[] wantMenus = {  
            new MenuDto(Long.valueOf(1), "Christmas menu",  
                new HashSet<DishDto>(  
                    Arrays.asList(new DishDto(Long.valueOf(1), "Turkey"), new DishDto(Long.valueOf(2), "Pecan Pie")))),  
            new MenuDto(Long.valueOf(2), "New year's eve menu", new HashSet<DishDto>(  
                Arrays.asList(new DishDto(Long.valueOf(3), "Potatos"), new DishDto(Long.valueOf(4), "Tiramisu"))));  
  
        // On fait la requête et on recupere la reponse.  
        ResponseEntity<MenuDto[]> response = this.template.getForEntity(getMenusURL().toString(), MenuDto[].class);  
  
        // On verifie le status de reponse.  
        assertEquals(HttpStatus.OK, response.getStatusCode());  
  
        // On list le corps de la reponse.  
        MenuDto[] gotMenus = response.getBody();  
    }  
}
```

Copy



Exercice: Activez les tests dans votre CI

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

- Les tests unitaires soient lancés
- Les tests d'integrations soient lancés



Pensez à bien regarder le cycle de vie des phases Maven

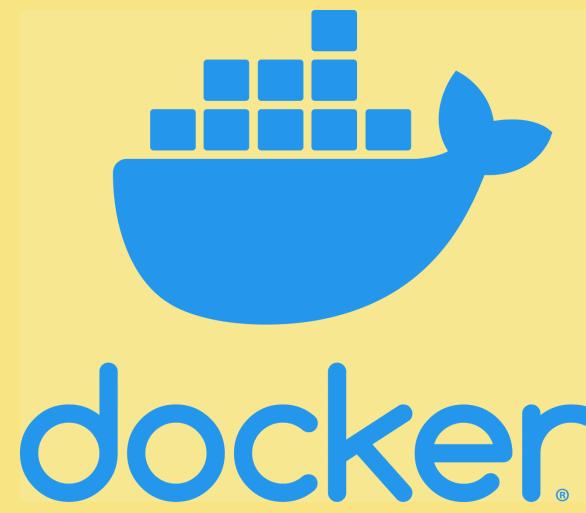
Checkpoint



On a vu :

- ✗ Les limites des tests unitaires
- 🏢 Comment faire des tests d'intégration
- 🤔 Tester n'est pas facile mais très utile

Docker



Remise à niveau / Rappels



Quel est le problème ?

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

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...

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?

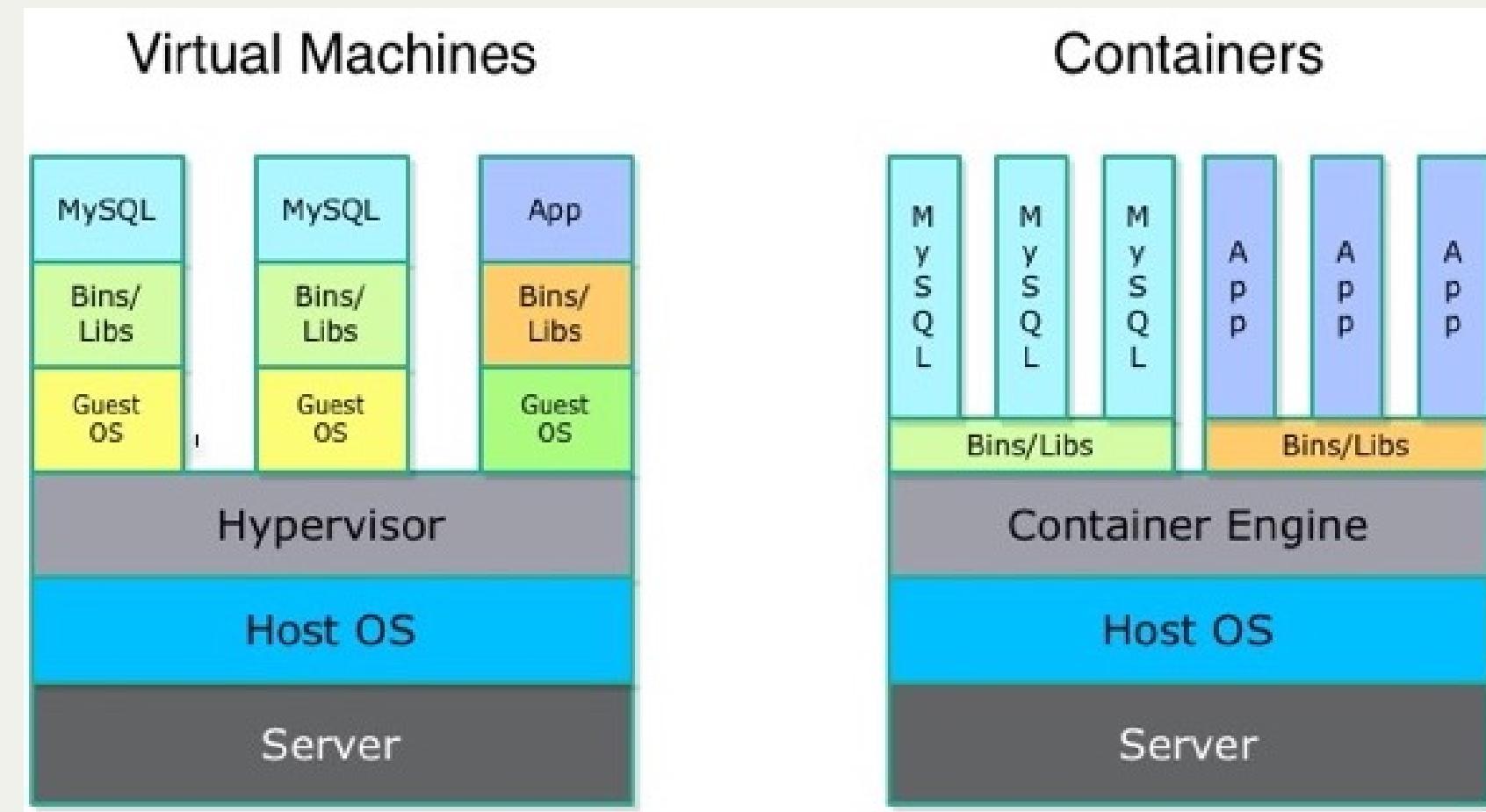
✓ Solution: Le conteneur intermodal

"Separation of Concerns"



Comment ça marche ?

"Virtualisation Légère"



Conteneur != VM

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

VM

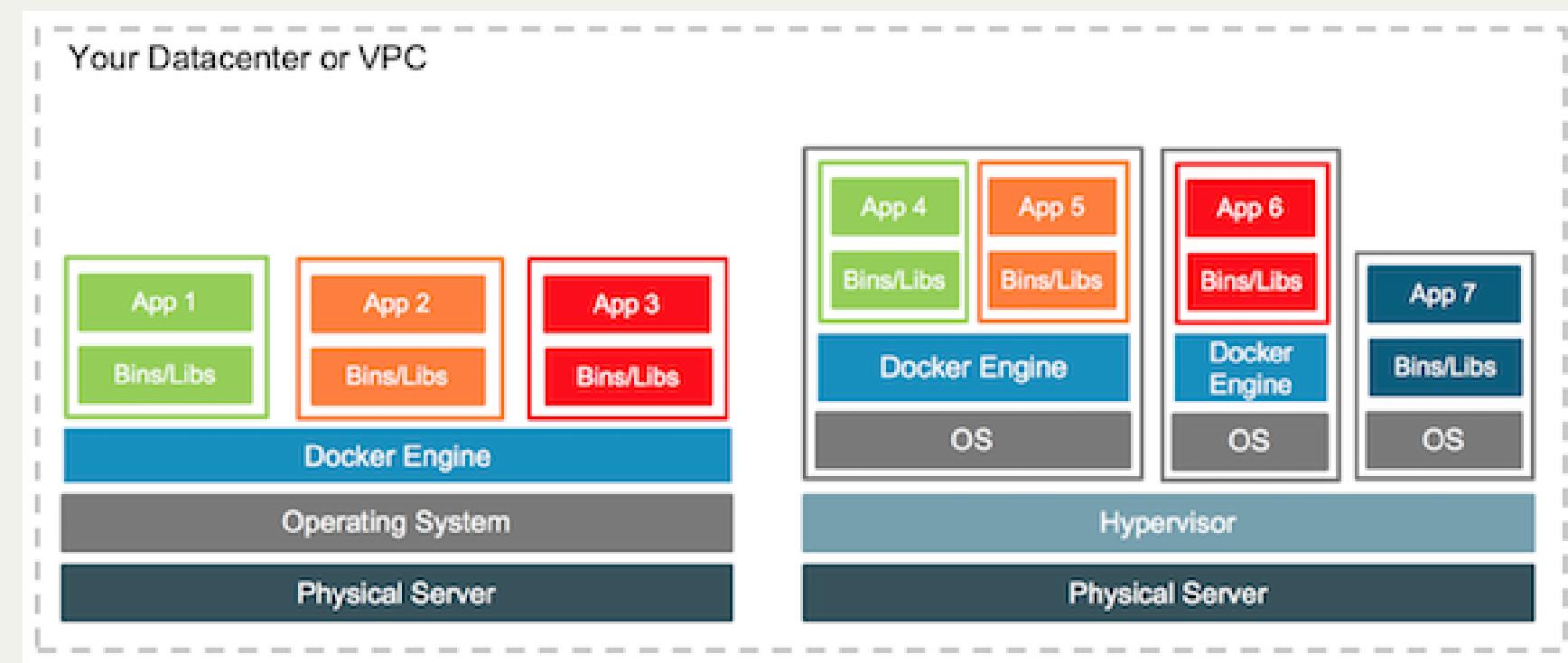


Containers



VMs && Conteneurs

Non exclusifs mutuellement





Exercice : où est mon conteneur ?

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

```
# 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 ?
```

Copy



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 téléchargé 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	Copy
109a9cdd3ec8	hello-world	"/hello"	33 seconds ago	Exited (0) 17 seconds ago		festive_faraday	

- 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**"



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

Copy

- ✓ 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
```

Copy



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 ?



Docker image layers

Shiny application

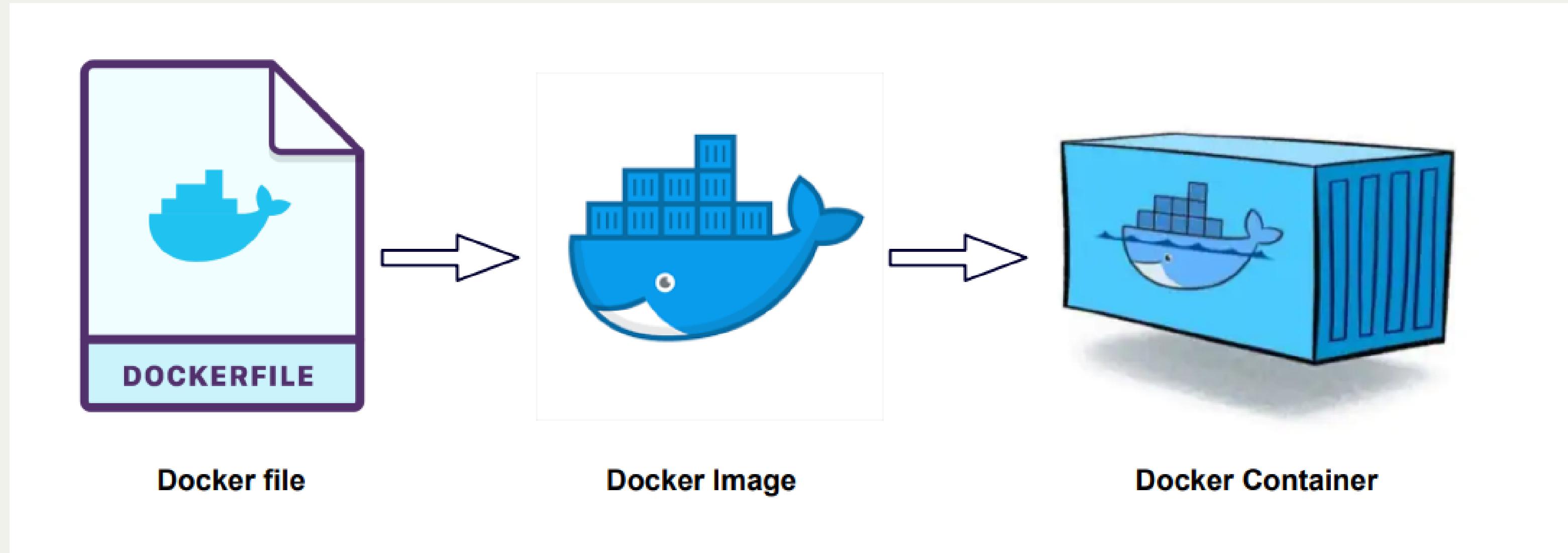
Build-time libraries & R package dependencies

Run-time system libraries

R & build tools

Base image: Ubuntu

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
 - \triangle 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-asciidoc ⇒
registry.docker.com/dduportal/docker-asciidoc:latest
- ghcr.io/dduportal/docker-asciidoc: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)

Copy

```
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
```

?

16 . 29

Checkpoint



- Une image Docker fournit un environnement de système de fichier auto-suffisant (application, dépendances, binaries, 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

Le logiciel que vous développez utilise des données d'une API d'un site de vente.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": 1343,
    // ...
  }
]
```

Copy

Voici comment est représenté un item vendu dans votre code.

```
public class Item {
  // Identifiant de l'item représenté sous forme d'entier.
  private int identifier;
  // ...
}
```

Copy



Le site décide tout d'un coup de changer le format de l'identifiant de son objet en chaîne de caractères.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": "lolilol13843",
    // ...
  }
]
```

Copy

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



com.fasterxml.jackson.databind.JsonMappingException



17.5

Qu'est s'est il passé ?

- Votre application ne s'attendait pas à un identifiant sous forme de chaîne de caractères !
- Le fournisseur de l'API a "changé le contrat" de son 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 manœuvre 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



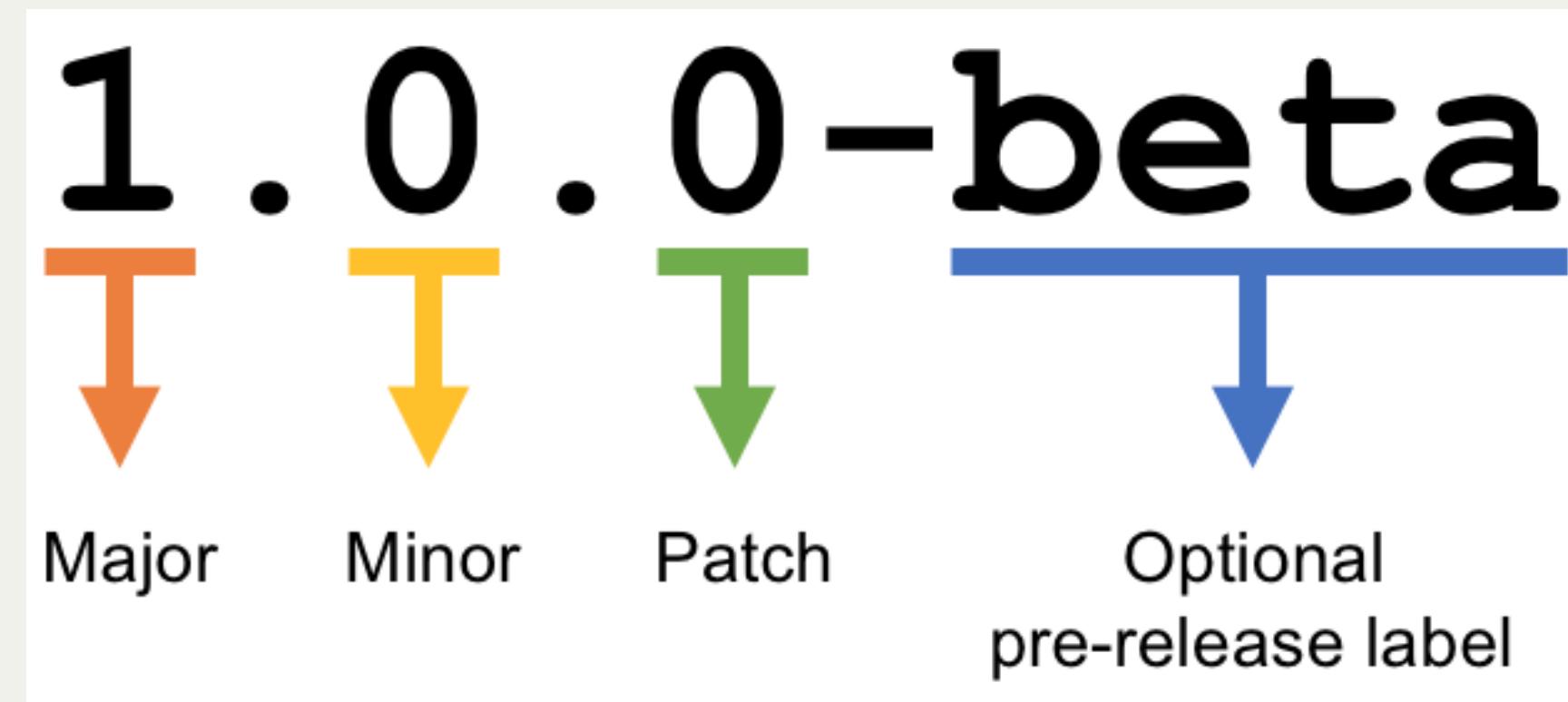
Quoi versionner ?

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 à l'utilisateur un moyen d'indiquer la version de l'API à laquelle il souhaite parler
 - Via un préfixe dans le chemin de la requête:
 - `https://monsupersite.com/api/v2.3/item`
 - 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.



Exercice : Créez et "taggez" la version v0.0.1 de votre menu-server

```
# Créer un tag.  
git tag -a v0.0.1 -m "Release version v0.0.1"
```

Copy

```
# Publier un tag sur le remote origin.  
git push origin v0.0.1
```

"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

Hello Github Releases!



Une release GitHub est associée à un tag git et porte :

- Un titre
- Un descriptif des changements
- Une collection de d'assets dont:
 - Des tarballs du code source à cette version (automatique)
 - Et éventuellement des fichiers de votre choix (des binaires compilés par exemple)

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`])

```
jobs:  
  release:  
    # Lance le job release uniquement si la branche est main.  
    if: contains('refs/heads/main', github.ref)  
    steps:  
      # ...
```

Copy



Exercice: créer une Release depuis le CI

Changez votre workflow de CI de façon à ce que sur un push de tag, le CI effectue les tâches suivantes dans un nouveau job:

- Une fois que l'étape build est terminée
- Télécharge et décomprime l'artefact généré par le job build
- Créer une nouvelle release dans votre dépôt ayant pour titre le nom du tag
- Upload jar de l'application dans cette release nouvellement créée

On vous suggère d'utiliser la CLI gh fournie par GitHub:

- Documentation de `gh release create`



Déploiement Continu

Continuous Deployment

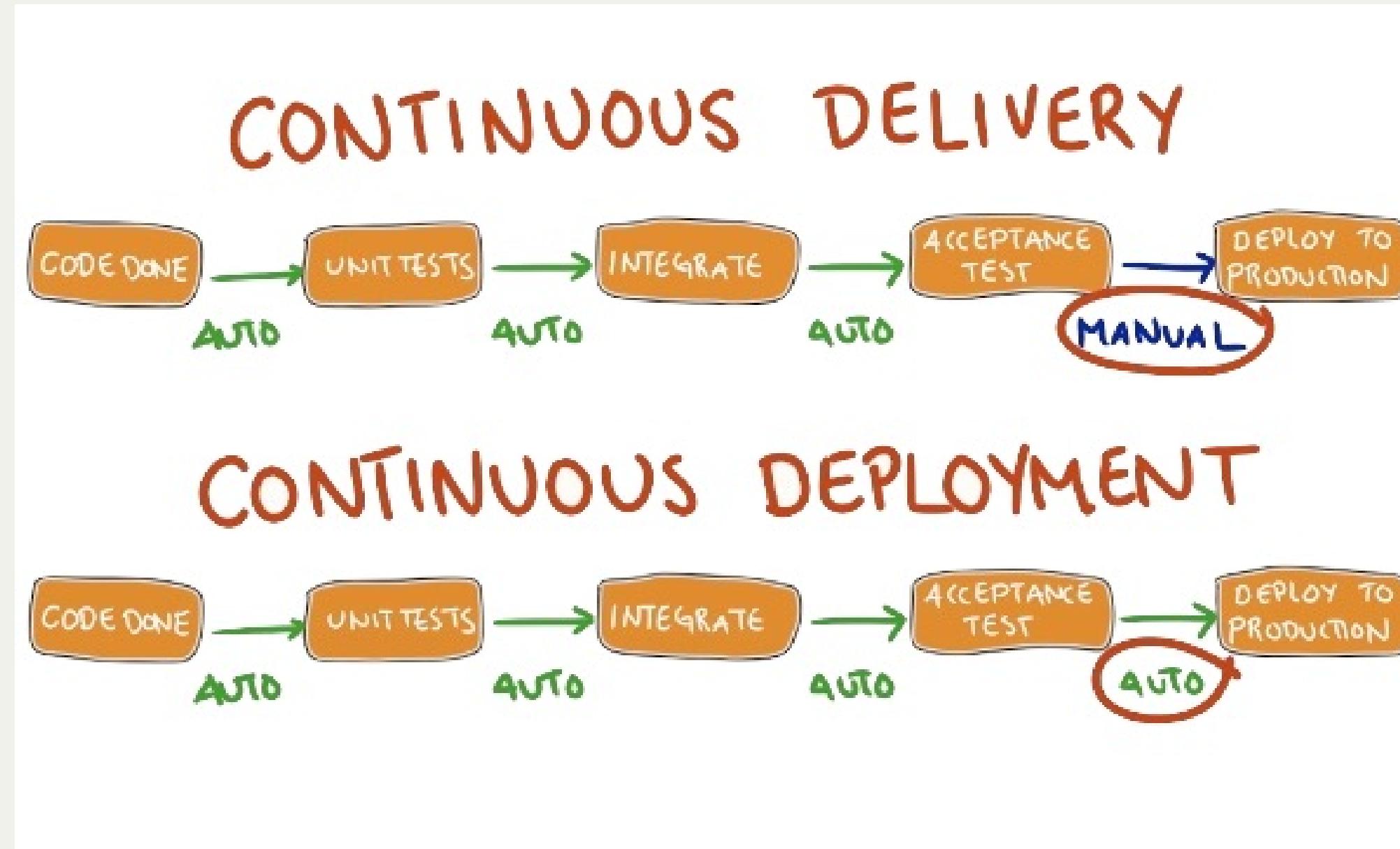




Qu'est ce que le Déploiement Continu ?

- 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...

Qu'est ce que "La production" ?

- Un (ou plusieurs) ordinateur ou votre / vos applications sont exécutées
- Ce sont là où vos utilisateurs "utilisent" votre code
 - Que ce soit un serveur web pour une application web
 - Ou un téléphone pour une application mobile
- Certaines plateformes sont plus ou moins outillées pour la mise en production automatique

Introduction à Google Cloud Run

- Dans le cadre de ce cours nous allons utiliser Google Cloud Run
 - Un Produit de Google Cloud Platform (GCP)
- Cloud Run Permet d'exécuter des images de containers ( wink wink) et de les exposer sur internet sans avoir à se soucier de l'infrastructure en dessous.
- Ideal dans le cadre de notre projet!
- Environnement sponsorisé par Voi pour le cours.

Construire une Image de Container pour Cloud Run

- A la racine de votre dépôt menu-server créez un fichier Dockerfile avec le contenu suivant :

```
# Depuis l'image de base azul/zulu-openjdk:11 (qui embarque un JRE dans la version 11)
FROM azul/zulu-openjdk:17

# Copier l'archive JAR depuis l'hôte dans le fichier /opt/app/menu-server.jar de l'image
COPY target/menu-server.jar /opt/app/menu-server.jar

# Définis la commande par défaut du container à java -jar /opt/app/menu-server.jar --server.port=${PORT}
# La variable d'environnement PORT est définie par Google Cloud Run à la création du container.
CMD ["java", "-jar", "/opt/app/menu-server.jar", "--server.port=${PORT}"]
```

Copy





Exercice: créez et exécutez l'image dans votre Workspace

- Dans un terminal, lancez les commandes suivantes:

```
# On repackage l'app
mvn package

IMAGE_NAME="cicdlectures/menu-server:test"

# Construit une image docker portant le tag `cicdlectures/menu-server:test`
docker image build --tag="${IMAGE_NAME}" ./

# Lance un container basé sur l'image `cicdlectures/menu-server:test` sans spécifier la variable d'environnement PORT
docker container run --tty --interactive --rm --publish 8080:9090 "${IMAGE_NAME}"
# Badaboum attendu!

# Lance un container basé sur l'image `cicdlectures/menu-server:test`
docker container run --tty --interactive --rm --env PORT=9090 --publish 8080:9090 "${IMAGE_NAME}"

# Vérifiez que vous pouvez faire des requêtes au menu-server....
# Et Ctrl+C pour terminer l'exécution du container
```

Copy

?

18.17

Déployer dans Cloud Run

Les grandes étapes d'un déploiement dans Cloud Run

1. On construit une image de container de l'application et la publie dans une registry d'images Google Cloud.
2. On demande ensuite à Cloud Run d'instancier un nouveau container utilisant la nouvelle image publiée.





Exercice: Connectez vous a Google Cloud depuis votre Workspace

Cela nécessite un compte Google, si vous n'en avez pas vous pouvez en créer un [ici](#).

```
# Authentifie votre instance gitpod auprès de google cloud
gcloud auth login

# Paramétrise le projet
GCP_PROJECT_NAME=voi-sdbx-ensg-2023
GCP_REGISTRY=europe-west9-docker.pkg.dev

# Indique a la CLI google cloud d'utiliser le projet partage pour le cours.
gcloud config set project "${GCP_PROJECT_NAME}"

# Authentifie le démon docker de votre instance auprès de la registry Google Cloud.
gcloud auth configure-docker "${GCP_REGISTRY}"
```

Copy





Exercice: Déployez votre Menu Server dans Cloud Run

```
GCP_IMAGE_NAME="${GCP_REGISTRY}/${GCP_PROJECT_NAME}/cicd-registry/<votre-binôme>/menu-server:test-cloudrun"
```

Copy

```
# On renomme l'image avec le nom de la registry  
docker image tag "${IMAGE_NAME}" "${GCP_IMAGE_NAME}"
```

```
# On publie l'image dans la registry google cloud  
docker image push "${GCP_IMAGE_NAME}"
```

```
# On déploie l'image publiée dans cloud run  
gcloud run deploy <votre-dépôt> --image="${GCP_IMAGE_NAME}" --region=europe-west9
```



Mais le faire manuellement c'est pas du déploiement continu! Il faut le faire depuis GitHub action!





Exercice: Mise en Place du Déploiement Continu dans Cloud Run depuis votre Workflow

Changez votre workflow de CI pour que, sur un évènement de push de tag de version:

- Une fois le build terminé un nouveau job `release-cloud-run` soit lancé
- Ce job effectue dans l'ordre:
 - Télécharge l'artefact de l'étape `build`
 - Checkout le depot (pour rapatrier le Dockerfile)
 - Appelle l'action `Cloud Run Release` pour déployer automatiquement une nouvelle version de votre application.

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%20%93eval%20%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/2023>



Source on : <https://github.com/cicd-lectures/slides>

