

# Introduction au CI/CD

ENSG - Décembre 2021

- Présentation disponible à l'adresse: <https://cicd-lectures.github.io/slides/2021>
- 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 : utiliser la touche "o" (pour "**Overview**")
- Pour voir les notes de l'auteur : utilisez la touche "s" (pour "**Speaker notes**")

# Bonjour !

# Damien DUPORTAL

- Señor 🌮 Software Engineer chez CloudBees sur le projet Jenkins 🎉
- Freelancer
- Me contacter :
  - ✉ damien.duportal <chez> gmail.com
  - 💬 Damien Duportal
  - 🐦 @DamienDuportal

# Julien LEVESY

- Senior Software Engineer @ Upfluence
- Me contacter :
  -  jlevesy <chez> gmail.com
  -  Julien Levesy
  -  @jlevesy
  -  @jlevesy

Et vous ?



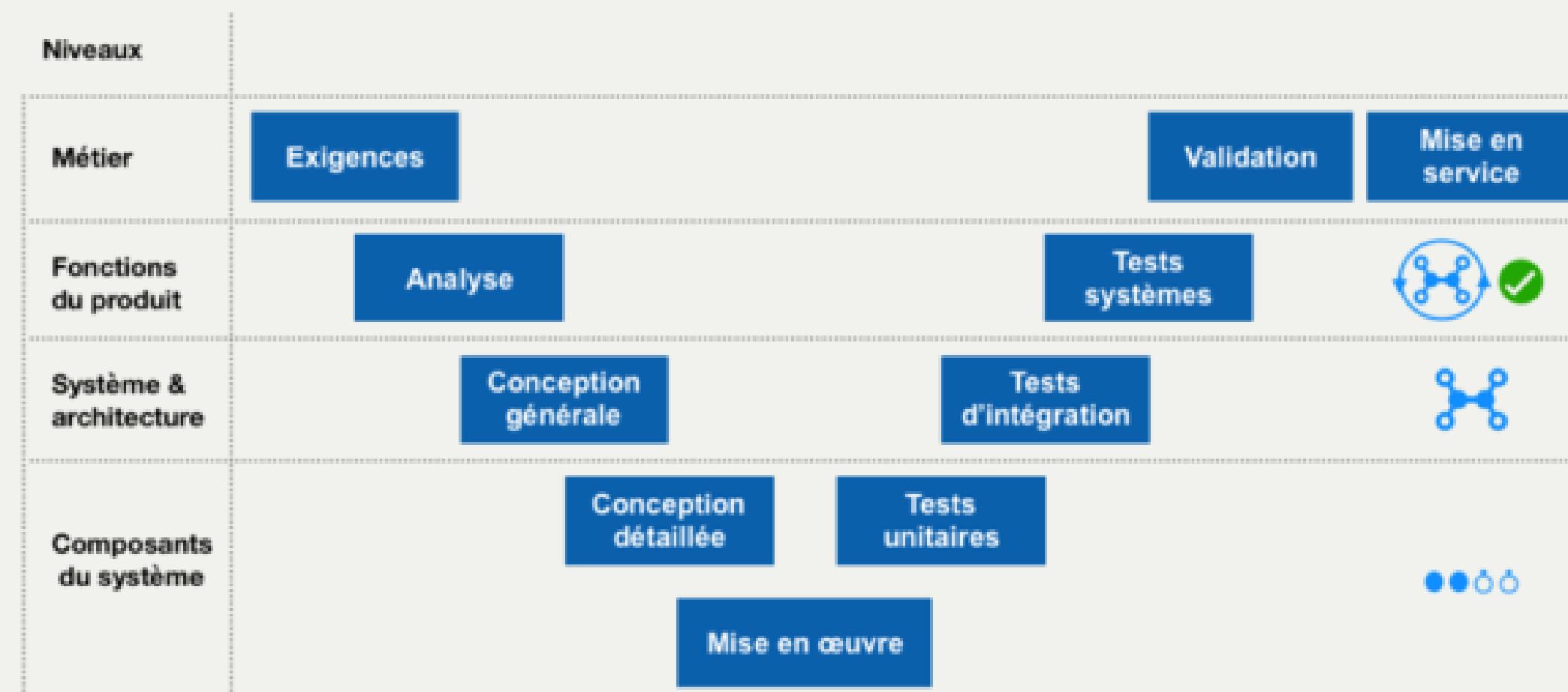
# A propos du cours

- 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



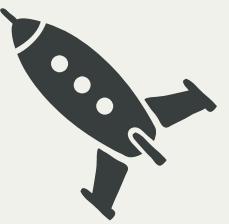
- Un navigateur récent (et décent)
- Un compte sur GitHub
- On va vous demander de travailler en binôme, commencez à réfléchir avec qui vous souhaitez travailler !

# 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

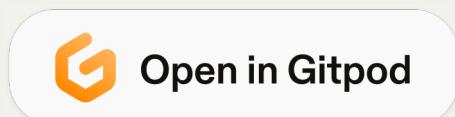
# Autorisations demandées par GitPod

Lors de votre 1er accès, GitPod va vous demander des autorisations (à accepter) sur votre compte GitHub:

- Accès à votre email public configuré dans GitHub (identification)
- Accès à vos dépôts GitHub public (accès aux dépôts dans l'éditeur)
- Accès aux GitHub workflows de vos dépôts publics (partie "CI" de ce cours)

# Démarrer l'environnement GitPod

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



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>

# 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▲
- Referez vous à la page [workspaces](#) pour accéder a la liste des environnements que vous avez créé.

# Checkpoint

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

On peut commencer !

# Les fondamentaux de git

# Tracer le changement dans le code

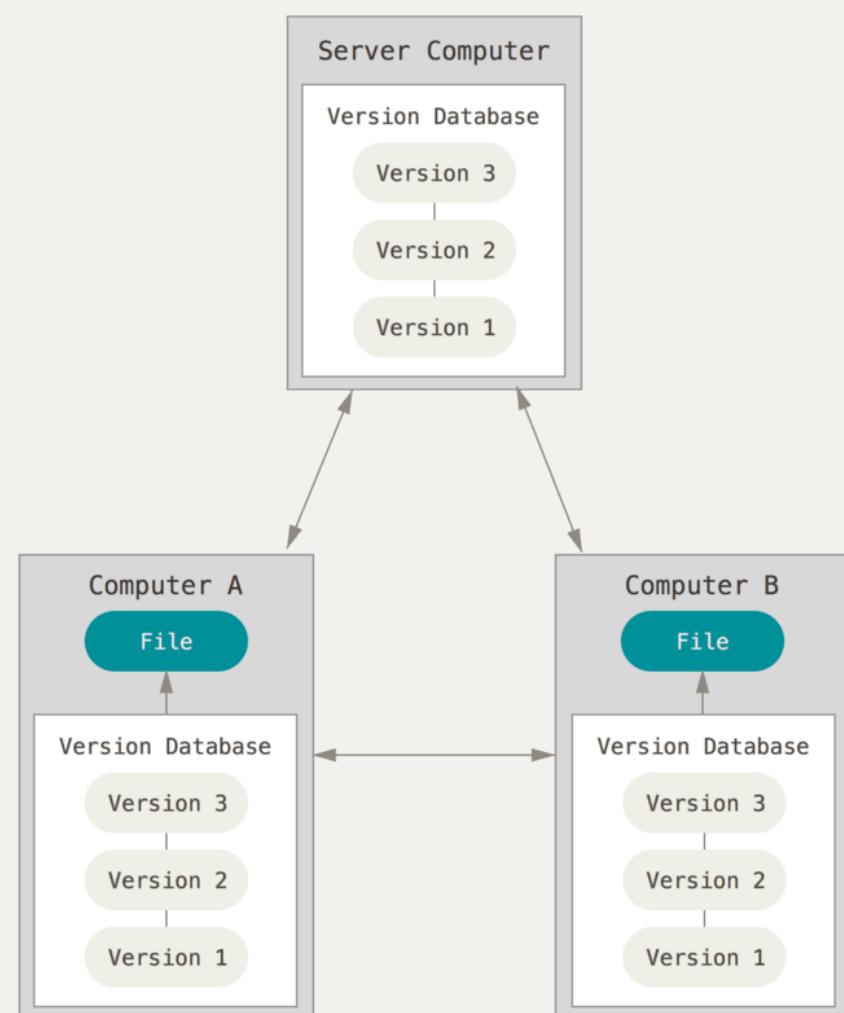
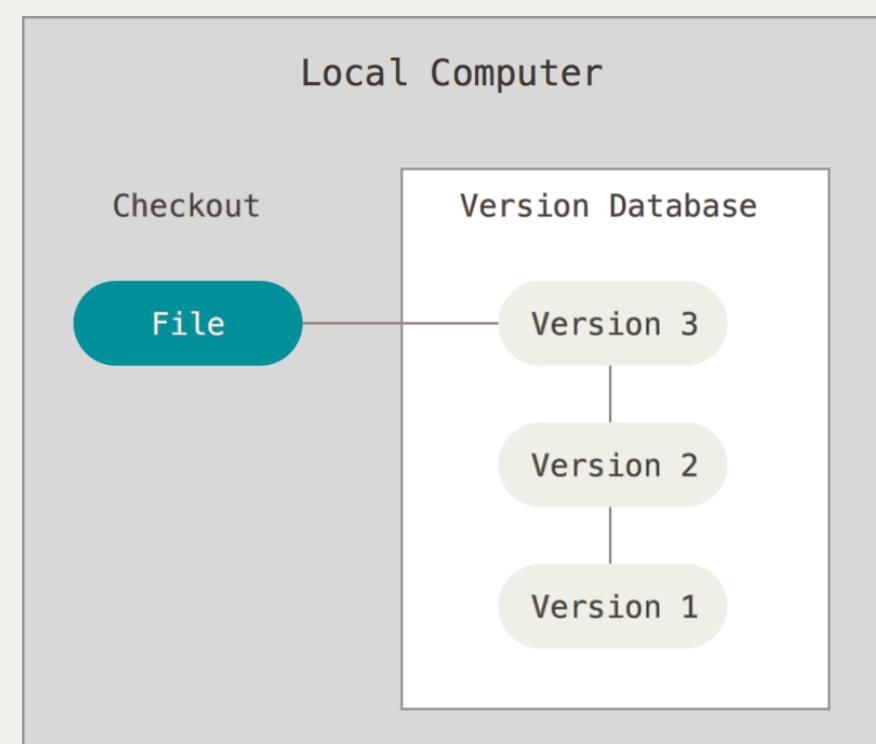
avec un **VCS** :  Version Control System

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

# Concepts des VCS



Source : <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

# Quel VCS utiliser ?



# Nous allons utiliser Git

# Git

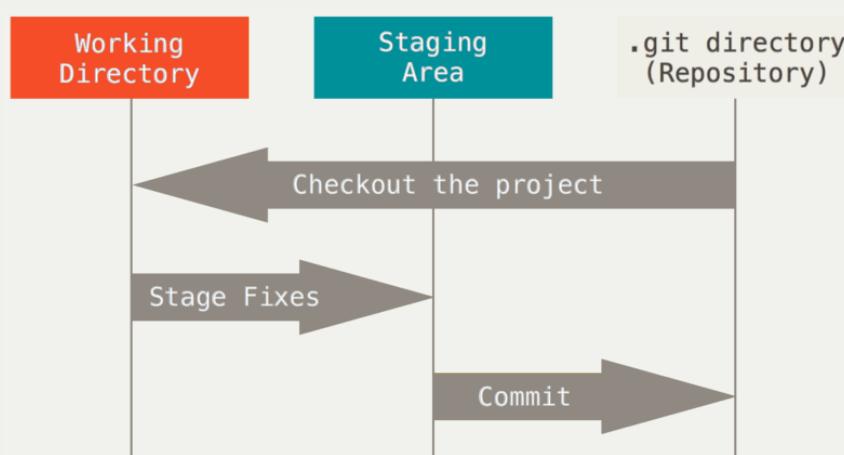
*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*

<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](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 de l'exercice 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 de l'exercice avec Git - 1.2

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

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

Copy

# Terminologie de Git - Diff et changeset

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

```
10 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: Recconcile
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: Recconcile
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 Jenkinsfile
- > 10 make.ps1
- > 456 tests/plugins-cli.Tests.ps1
- > 2 tests/plugins-cli.bats
- > 2 tests/plugins-cli/Dockerfile-windows
- > 16 tests/plugins-cli/custom-war/Dockerfile-windows

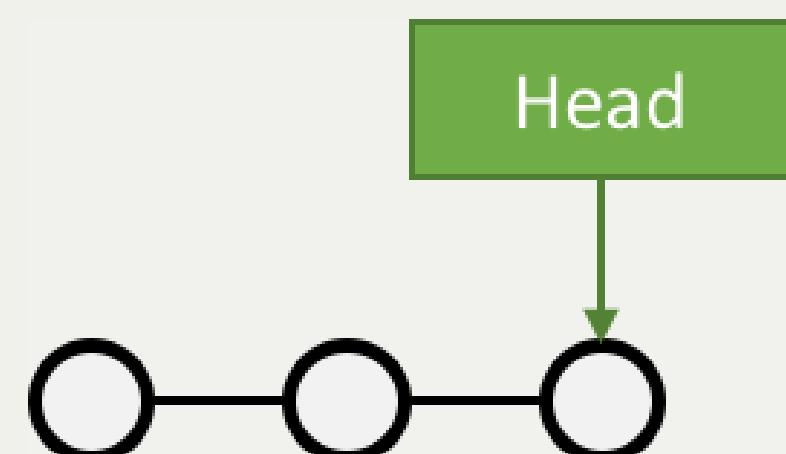
# Terminologie de Git - Commit

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

The screenshot shows a GitHub commit page for a single commit on the master branch. The commit message is "Bump node-problem-detector to v0.8.5". It was made by user "tos13k" 2 days ago. The commit has one parent, commit e64ebe0, and its SHA-1 is 8f2dd3aaab02c3b4c1c8233aa5f93bb439f23228. Below the commit details, it says "Showing 3 changed files with 8 additions and 8 deletions." There are "Unified" and "Split" options for viewing the diff.

**"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 de l'exercice avec Git - 2

```
git log
```

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

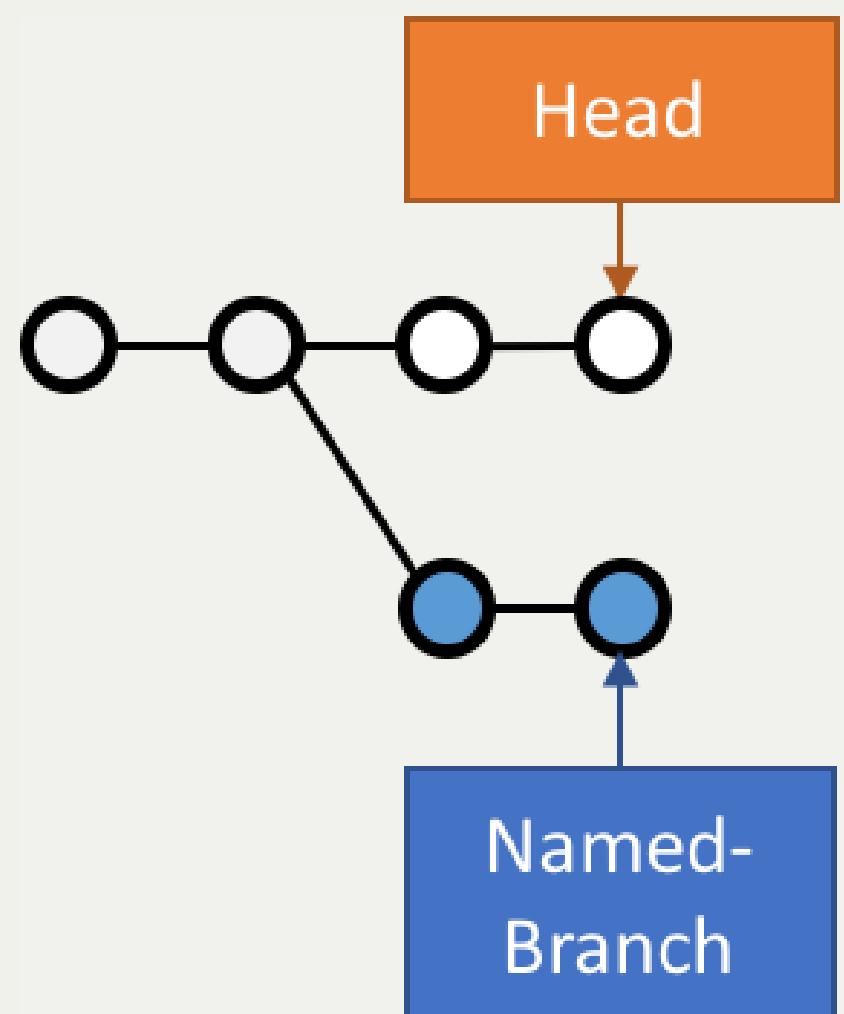
```
git diff  
git status
```

```
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 de l'exercice avec Git - 3

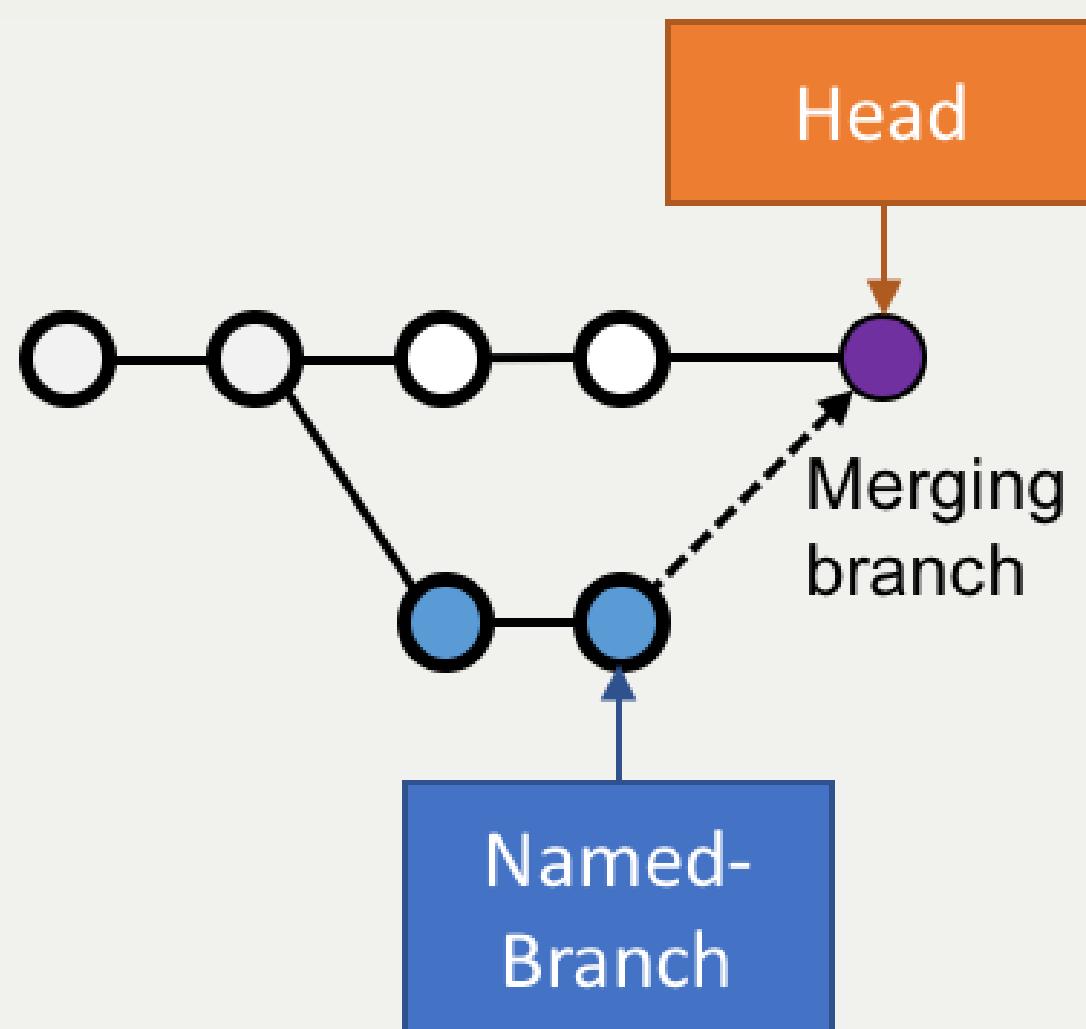
```
git branch feature/html && git checkout feature/html
# Ou git checkout -b feature/html
echo '<h1>Hello</h1>' > ./index.html
git add ./index.html && git commit -m "Ajout d'une page HTML par défaut"

git log --graph
# git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset'
```

Copy

# Terminologie de Git - Merge

- On intègre une branche dans une autre en effectuant un **merge**
  - 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`
- Afficher le graphe correspondant à cette branche avec `git log --graph`

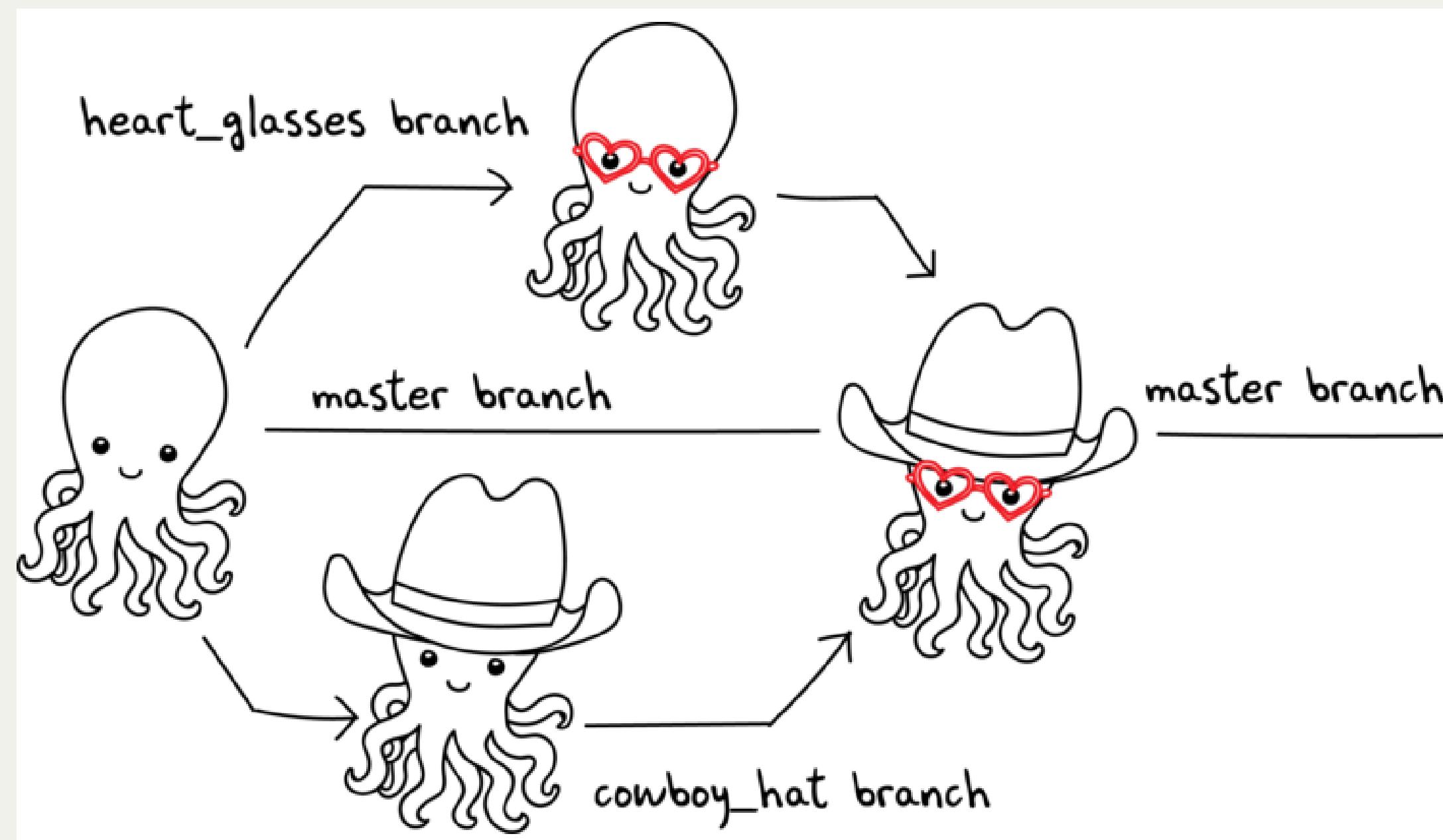
# Solution de l'exercice avec Git - 4

```
git checkout master
git merge --no-ff feature/html
git log --graph
# git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset'
```

Copy

# Feature Branch Flow

- **Une seule branche par fonctionnalité**



# 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>

# Pour aller plus loin avec Git et les VCS...

Un peu de lecture :

- <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/>

# Présentation de votre projet

# Contexte Géopolitique (1/2)

- Dans un effort d'optimisation de ses processus métiers, la cantina de Mos-Estafette s'est lancée dans un grand projet de digitalisation de la gestion de sa carte
- Le projet est divisé en deux lots:
  - Une partie serveur qui stocke et expose la données de la carte via une API REST
  - Une partie client qui permet de consulter et de gérer les menus de la cantina via l'API du serveur

# Contexte Géopolitique (2/2)

- Après une longue et épique bataille de chefs de projets et autres décideurs fonctionnels, la société **Bananes Services d'Entreprises**, spécialiste national des technologies du minitel, à remporté le lot de la partie serveur...
- ...Cependant, suite à d'un **grave accident de brainstorming** entre les 18 chefs de projet travaillant sur le dit projet et 6 mois de retard sur la première livraison, l'entreprise **Bananes** s'est retrouvée dans l'incapacité de mener le projet à son terme
- Désemparés face à cette terrible nouvelle, le conseil d'administration de la cantina à décidé de se tourner vers l'ENSG pour tenter de sauver leur transition numérique

# 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/2021/media/menu-server.tar.gz

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

Copy

A partir de la vous pouvez ouvrir le fichier LISEZMOI.txt et commencer à suivre ses instructions.

# Comment accéder à un serveur démarré dans un Gitpod ?

- Vous avez probablement démarré un serveur web qui écoute sur le port 8080
- Si vous souhaitez accéder par `curl` depuis terminal de votre gitpod, rien à faire
- Si vous souhaitez y accéder depuis l'extérieur de votre gitpod il vous faut une URL publique:
  - `gp url 8080` vous indique l'URL du port 8080 de votre instance gitpod!
  - `gp preview "$(gp url 8080)"` ouvrir votre navigateur.

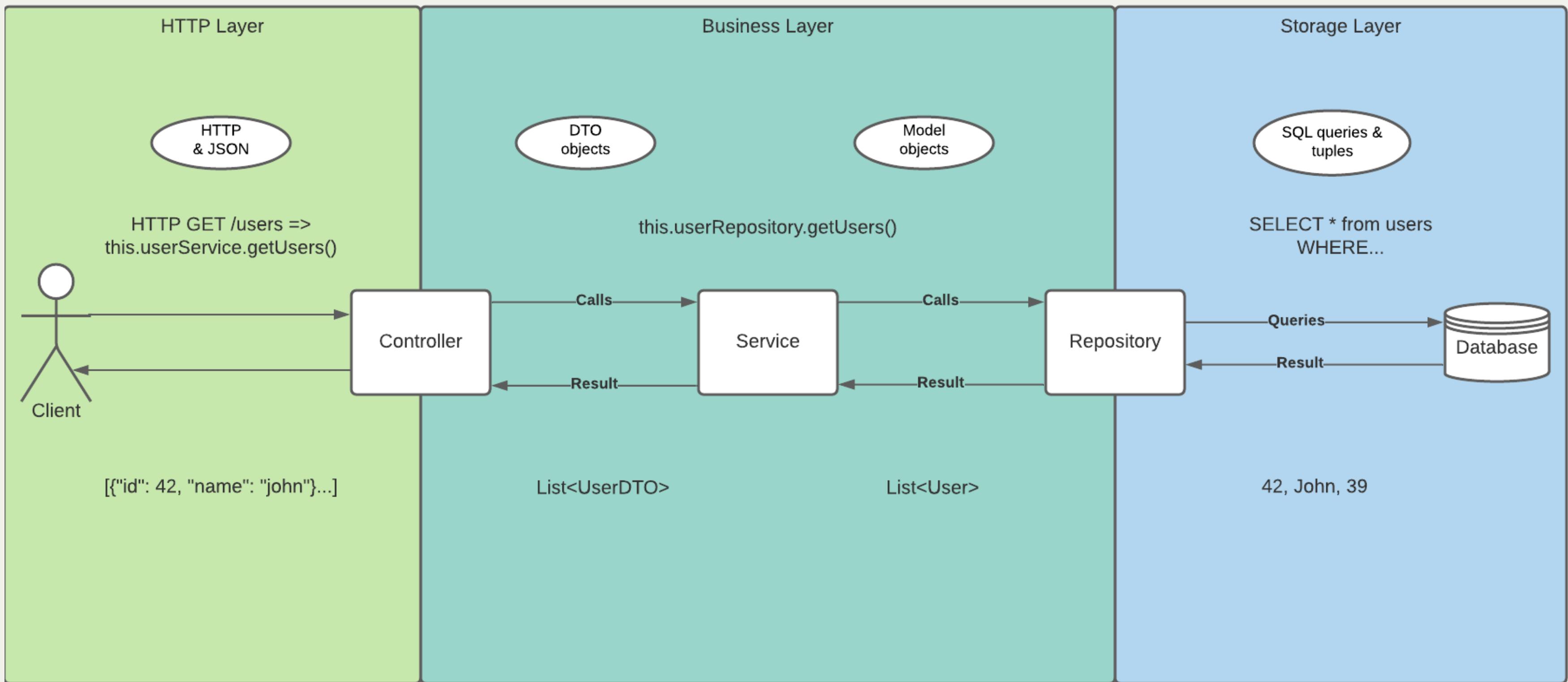
Qu'est-ce qui va / ne va pas dans ce projet  
d'après vous?

# 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!

# Architecture du projet



# HTTP Layer

- Fait la transition entre une requête HTTP et une action logique de l'application
- Accepte en entrée une requête HTTP (headers, query parameters, verb, (JSON) body)
- Réponds un code de statut, des headers et optionnellement un body
- Appelle la couche service en passant des DTOs (data transfer objects)

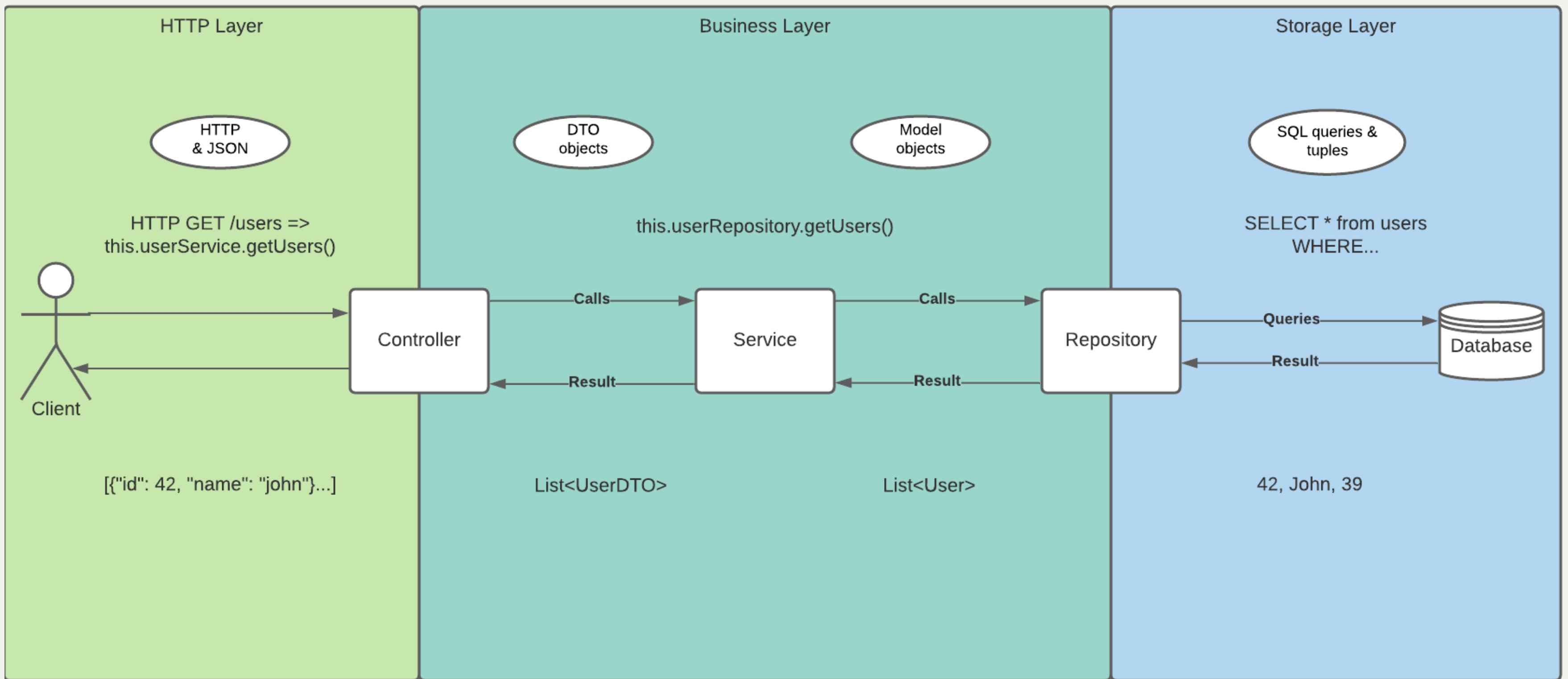
# Business Layer

- Implémente la logique métier de l'application
- Accepte en entrée des DTOs et en réponds en sortie
- Appelle d'autre services ou les repositories de l'application en leur passant des objets model

# Storage Layer

- Fait la transition entre le modèle objet java et le modèle de la base de données (ici relationnel // SQL)
- Accepte en entrée des objets du modèle, et peut en répondre aussi
- Chaque méthode implémentée par un repository est en réalité une requête à votre base de données
- Implémentation ici déléguée à un outil d'ORM (object relational mapping), ici hibernate
- On utilisera dans le cadre de ce cours une base de donnée in-memory appelée **h2**

# Architecture: résumé



# Modèle de données

- Une entité Menu, portant un identifiant et un titre
- Une entité Dish, portant un identifiant et un titre
- Un Menu est composé de 0 à N Dish

# Organisation du Code Source

Toutes les classes sont créées dans un sous répertoire de  
src/main/java/com/cicdlectures/menusever

Le découpage est le suivant:

- controller: Couche d'interconnection entre HTTP et le domaine métier
- dto (Data Transfer Objects): Représentation intermédiaire de la donnée
- service: Couche métier
- repository: Couche d'accès a la base de données
- model: Représentation de votre modèle de donnée, configuration de l'ORM.

# 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

# Solution Exercice

```
# On évacue le contenu inutile
rm -rf dist/
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

# Cycle de vie de votre projet

# Quel est le problème ?

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

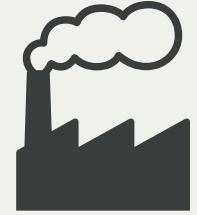
- Qu'est ce qu'on "fabrique" à partir du code ?
- Comment faire pour "fabriquer" de la même manière pour tout•e•s (💻 | 💻 ) ?

# 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*

# Reproduire la fabrication



Comment fabriquer et tester de manière reproductible ?

- Dimension technique: type de language, différents OS, différentes machines
- Dimension temporelle: version de dépendances qui changent dans le temps
- Dimension humaine: habitudes, connaissances, documentation (ou pas)

# Exemple avec Java ☕

JAVA bien ?

- Essayons un projet "Hello World" en Java dans GitPod, dans un nouveau dossier vide :



# Exemple Java : Compiler

- Créez un fichier source java main.java avec le contenu suivant :

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello ENSG!");  
    }  
}
```

Copy

- La commande javac permet de "compiler" du java en bytecode java.
  - Compilez le programme dans un dossier target / :

```
javac ./main.java -d ./target  
ls -ltra ./target # Quel est le résultat ?
```

Copy

# Exemple Java : Exécuter (Artisanalement)

- Un fichier `HelloWorld.class` a été généré
  - C'est du "bytecode binaire" Java exécutable avec la commande `java`
- Essayez d'exécuter le programme :
  - La commande `java` a besoin de connaître le dossier contenant les classes

```
java -classpath ./target/ HelloWorld
```

Copy

# Exemple Java : Livrable JAR



- 🤔 Il faut connaître le nom des classes à exécuter ?
- Réponse : Fichier "JAR" (Java ARchive)
  - C'est une archive "ZIP" pour distribuer des applications en java
  - Pour exécuter le programme depuis le livrable :

```
java -jar <fichier.jar>
```

Copy

# Exemple Java : Fabriquer un JAR (Artisanalement)



- Il faut un fichier descripteur appelé MANIFEST.mf :

```
Main-Class: HelloWorld  
Class-Path: target/
```

[Copy](#)

- Il faut utiliser la commande jar pour fabriquer l'archive puis la tester :

```
jar --create --manifest=MANIFEST.mf --file helloworld.jar target/HelloWorld.class  
java -jar helloworld.jar
```

[Copy](#)

# Exemple réel "à la main" Java → JAR

- Essayons avec un vrai projet : reprenons le projet menu-server
- Essayez de le compiler avec javac :

```
javac ./src/main/java/com/cicdlectures/menuserver/MenuServerApplication.java -d ./target
```

```
# ...
./src/main/java/com/cicdlectures/menuserver/MenuServerApplication.java:3: error: package org.springframework.boot does not exist
import org.springframework.boot.SpringApplication;
^
# ...
```

## Problème de dépendances

# Checkpoint

⇒ C'est vite **complexe** alors qu'on veut fabriquer / tester un programme

# Do It Ourselves or Reinvent the Wheel ?

## Problème :

Doit-on redéfinir tout ce processus à chaque fois ?



## Réponse(s) :

- Dans la vraie vie, ça dépend donc ce sera à vous de juger et de ré-évaluer
- Pour ce cours: 🔨 il faut un outil pour gérer le cycle de vie technique (Build → Test → etc.)

# Introduction à Maven

Say Hello to "Maven":

- Idée de Maven : **Cycles de vie** standardisés
- "Convention over configuration" : fichier `pom.xml` décrivant le projet
- Ligne de commande `mvn` qui lit le `pom.xml` et exécute les **phases**

# Maven : Cycles de vie

- 3 cycles de vie :
  -  default : pour construire les applications dans target /
  -  site : pour construire un site web de documentation dans target /
  -  clean : nettoyer target /
- Chaque cycle de vie est composé d'une série d'étapes appelées "phases"

# Maven : Phases

Phases du cycle de vie  default :

- validate - Validation du projet (syntaxe du pom.xml et du Java, etc.)
- compile - Fabrication des fichiers .class depuis le code java
- test - Exécuter les tests unitaires
- package - Préparer le livrable finale (.jar par exemple)
- verify - Exécuter les tests d'intégration
- install - Mettre à disposition le livrable localement pour d'autres projets Maven
- deploy - Copier le livrable dans un système de stockage de dépendance distant

# Maven : pom.xml

- "POM" signifie "Project Object Model"
- Contenu en XML : language de type "markup", avec un **schéma**, donc strict
- "Convention au lieu de configuration" pour limiter la complexité
  - code source attendu dans `src/main/java/`
  - résultats dans `target/` (transient), etc.
  - `pom.xml` à la racine de votre projet

# Maven : Ligne de commande mvn

Ligne de commande mvn :

- Lit le fichier pom.xml pour comprendre le projet
- Attend une (ou plusieurs) phases en argument
- Accepte des options (formes courtes -X ou longues --debug)

```
mvn clean # Appelle la phase "clean"  
mvn compile # Appelle les phases "validate" puis "compile"  
mvn clean compile -X # On peut appeler plusieurs phases et passer des options
```

Copy

# Exercice Maven : C'est à vous !

**But :** fabriquer l'application menuserver avec Maven

 Open in Gitpod

Commençons par valider le projet en utilisant la phase validate de Maven:

```
mvn validate
```

Copy

```
# ...
[ERROR] The goal you specified requires a project to execute but there is no POM in this directory (/workspace/menu-serve
# ...
```

Copy

✖ Il manque un fichier pom.xml !

# Exercice Maven : fichier pom.xml

- Commençons par créer un fichier pom.xml avec le contenu ci-dessous :

```
<!-- pom.xml -->
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- Insert content here -->
</project>
```

Copy

- Puis ré-essayons de valider le projet avec Maven :

```
mvn validate
```

Copy

```
# ...
[ERROR] [ERROR] Some problems were encountered while processing the POMs:
[FATAL] 'groupId' is missing. @ line 2, column 102
[FATAL] 'artifactId' is missing. @ line 2, column 102
[FATAL] 'version' is missing. @ line 2, column 102
# ...
```

Copy

✖ On doit ajouter du contenu dans le pom.xml !

# Maven : identité d'un projet

Maven identifie un projet avec les 3 éléments **obligatoires** suivants :

- **groupId** : Identifiant unique de votre projet suivant les règles Java de nommage de paquets
- **artifactId** : Identifiant du projet (paquet de la classe principale)
- **version** : Version de l'artefact

```
<!-- pom.xml -->
<groupId>com.mycompany</groupId>
<artifactId>my-app</artifactId>
<version>1.0</version>
<!-- Example avec un paquet Java `com.mycompany.my-app` dans `src/main/java/com/mycompany/my-app` -->
```

Copy

# Exercice Maven : identifiez votre projet

⇒ C'est à vous

- Identifiez votre projet en remplissant le fichier pom.xml
  - groupId et artifactId: utilisez le nom de package de votre classe principale MenuServerApplication.java
  - version : 1.0-SNAPSHOT
- **Objectif :** Maven doit valider le projet avec succès :

```
mvn validate
```

Copy

```
# ...
[INFO] BUILD SUCCESS
# ...
```

Copy

# Checkpoint

- On a pu créer un fichier pom.xml valide ✓
- Pensez à commiter ce changement 🍌
- Il est temps de compiler l'application avec Maven 

# Exemple Maven : Compiler



- Essayez de compiler l'application à l'aide de la phase `compile` de Maven:

```
mvn compile
```

Copy

- Résultat attendu : Message [INFO] BUILD FAILURE ✘

# Analyse des erreurs de compilation



Que s'est il passé ?

1. ⇒ Maven a téléchargé plein de dépendances depuis <https://repo.maven.apache.org>

2. ⇒ La compilation a échoué avec 2 erreurs et 1 warning :

- ✗ [ERROR] Source option 5 is no longer supported. Use 6 or later.
- ✗ [ERROR] Target option 1.5 is no longer supported. Use 1.6 or later.
- ⚠ [WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!

# Maven et Dépendances Externes

- Maven propose 2 types de dépendances externes :
  - **Plugin** : c'est un artefact qui sera utilisé par Maven durant son cycle de vie
    - "Build-time dependency"
  - **Dépendance** ( "dependency") : c'est un artefact qui sera utilisé par votre application, *en dehors de Maven*
    - "Run-time dependency"

# Maven et Plugins

Quand on regarde sous le capot, Maven est un framework d'exécution de plugins.

⇒ Tout est plugin :

- Effacer le dossier target ? Un plugin ! (si si essayez mvn clean une première fois...)
- Compiler du Java ? Un plugin !
- Pas de plugin qui fait ce que vous voulez ? Écrivez un autre plugin !

C'est bien gentil mais comment corriger l'erreur

**X Source** option 5 is no longer supported. Use 7 or later ?

- C'est le maven-compiler-plugin qui a émis l'erreur
- Que dit la documentation du plugin ?
  - Also note that at present the default source setting is 1.6 and...
- Il faut définir la version de Java à utiliser pour le programme (e.g. celle en **production**)

# Maven Properties

- Maven permet de définir des propriétés (🇬🇧 "properties") "CLEF=VALEUR" pour :
  - Configurer les plugins (📦)
  - Factoriser un élément répété (une version, une chaîne de texte, etc.)
- Le fichier pom.xml supporte donc la balise <properties></properties> pour définir des propriétés sous la forme <clef>valeur</clef> :
  - La propriété peut être utilisé sous la forme \${clef}

```
<properties>
  <spring.version>1.0.0</spring.version>
  <ensg.student.name>Damien</ensg.student.name>
</properties>

<build>
  <name>${ensg.student.name}</name>
</build>
```

Copy

# Exercice : Corriger le "warning"

⇒ C'est à vous !

- **But :** Résoudre le message de "Warning" à propos de l'encodage des fichiers
  - On veut forcer l'encodage à UTF-8 pour que les caractères accentuées soient supportés de manière portable
- Modifiez le fichier pom.xml pour ajouter un bloc <properties> pour définir l'encodage à UTF-8 :
  -  FAQ Maven ("How do I prevent encoding warnings ?")

# Solution : Corriger le "warning"

```
<!-- pom.xml -->
<!-- ... -->
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<!-- ... -->
```

Copy

# Exercice : Corriger les 2 erreurs

- **But :** Résoudre les 2 erreurs de compilation à propos de la version de Java
- Modifiez le bloc <properties> du fichier pom.xml pour définir les versions de java pour les sources ET l'exécution :
  - 🔎 documentation du Maven Compile Plugin
  - 🔎 Obtenir la version de java dans GitPod : `java -version`
- Résultat attendu : **nouvelles** erreurs de compilation

# Solution : Corriger les 2 erreurs

```
<!-- pom.xml -->
<!-- ... -->
<properties>
    <!-- ... -->
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
<!-- ... -->
```

Copy

# Checkpoint

- On a pu éditer le fichier pom.xml pour décrire nos sources java ✓
  - Pensez à commiter ce changement 🎉
  - Il manque des dépendances pour compiler :
- ✗ package org.springframework.beans.factory.annotation does not exist

# Dépendances Externes

**Hypothèse :** on a besoin de code et d'outils externes (e.g. écrits par quelqu'un d'autre)

- Comment faire si le code externe est mis à jour ?
  - <https://github.blog/2020-11-16-standing-up-for-developers-youtube-dl-is-back/>
- Que se passe t'il si le code externe est supprimé de l'internet ?
  - <https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/>
- Acceptez-vous d'exécuter le code de quelqu'un d'autre sur votre machine ?
- Et si quelqu'un injecte du code malicieux dans le code externe ?
  - <https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/>

# TOUS les languages...

... sont concernés

# Maven : Dépôts d'Artifacts

Maven récupère les dépendances (et plugins) dans des dépôts d'artefacts  
(🇬🇧 Artifacts Repositories) qui sont de 3 types :

- **Central** : un dépôt géré par la communauté - <https://repo.maven.apache.org>
  - Avec une interface web de recherche
- **Remote** : des dépôts privés de votre organisation
- **Local** : un dossier sur la machine où la commande mvn est exécuté, généralement dans $\$\{HOME\}/.m2$ 
  - `mvn install` cible ce dépôt "local"

# Dépendances Maven

 "Introduction au mécanisme de dépendances - documentation Maven

- Pour spécifier les dépendances (dans votre pom.xml):
  - Il faut utiliser la balise <dependencies>,
  - ... qui est une collection de dépendances (balise <dependency> - quelle surprise !),
  - .. chaque dépendance étant défini par un trio <groupId>, <artifactId> et <version> (que de surprises...)
- Pour les plugins c'est la même idée (<plugins> → <plugin> → <groupId>, <artifactId>, <version>)

# Exemple de Dépendance : Spring

- **Idée :** Nous avons besoin d'ajouter le framework Spring en dépendance.

Voilà ce que ça donne dans le fichier pom.xml :

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.6.0</version>
  </dependency>
</dependencies>
```

Copy

# Exercice avec les dépendances Spring 1/2

⇒ C'est à vous.

- Ajoutez le bloc <dependencies> de la slide précédente dans votre pom.xml
  - 💡 org.springframework.boot:spring-boot-starter-web 2.6.0 sur Maven Central
- Exécutez la commande mvn compile
- Résultat attendu : **nouvelles ✖ erreurs de compilation** (encore une dépendance manquante) :

```
[ERROR] COMPILATION ERROR :  
[INFO] -----  
[ERROR] <...> package org.springframework.transaction.annotation does not exist
```

Copy

# Exercice avec les dépendances Spring 2/2

- **But:** Compiler l'application complète
- Continuez de modifier le fichier pom.xml afin d'ajouter les 2 dépendances suivantes :
  - org.springframework.boot:spring-boot-starter-data-jpa 2.6.0 sur Maven Central
  - org.projectlombok:lombok 1.18.22 sur Maven Central
- Résultat attendu : ✓ [INFO] BUILD SUCCESS

# Solution avec les dépendances Spring

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cicdlectures</groupId>
  <artifactId>menuserver</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.22</version>
    </dependency>
  </dependencies>
```

Copy

# Exécution de l'application Spring : Tentative 1

- Quel est le contenu de target/ ? Et de target/classes ?
  - `find ./target/classes -type f`
- Essayons d'exécuter notre programme avec la commande java :

```
java -cp target/classes/com/cicdlectures/menuserver/ \
      -cp target/classes/com/cicdlectures/menuserver/controller/ \
      -cp target/classes/com/cicdlectures/menuserver/dto/ \
      -cp target/classes/com/cicdlectures/menuserver/model/ \
      -cp target/classes/com/cicdlectures/menuserver/repository/ \
      -cp target/classes/com/cicdlectures/menuserver/service/ \
      MenuServerApplication
```

Copy

## Résultat :

```
Error: Could not find or load main class MenuServerApplication  
Caused by: java.lang.ClassNotFoundException: MenuServerApplication
```

Copy



# Checkpoint

- mvn compile a produit des fichiers dans target/classes/\*\* ✓
- C'est la galère pour trouver les bonnes dépendances 😞
- Il faut encore pouvoir exécuter l'application

⇒ reprenons en lisant le documentation

# Spring Boot : Read The Manual

- Spring Boot est bien plus simple à utiliser que ce que l'on a vu !
  - On l'a abordé ainsi pour mieux comprendre
- Une documentation très complète :
  - "Get Started" pour bien démarrer
  - Spring Initializr pour générer son pom.xml en ligne
  - Une documentation de référence
- Un plugin Maven est fourni par le projet Spring Boot pour se simplifier la vie:
  - Pas besoin de répéter les versions
  - Plein de fonctionnalités de développement
  - Moins de configuration à faire soit même

# Maven Plugins

Un plugin Maven implémente les tâches à effectuer durant les différentes phases, et peut appartenir à l'un ou l'autre de ces 2 types :

- "**Build**" : Implémente une action durant les phase du cycle de vie default, et est configuré dans la balise <build>
- "**Reporting**" Implémente une action durant les phases du cycle de vie site, et est configuré dans la balise <reporting> (à votre grande surprise)

C'est un fichier \*.jar identifié par... groupId, artifactId et version.

# Exemple Maven : plugin Spring Boot 1/2

⇒ C'est à vous !

- **But :** Configurer le plugin Spring Boot pour Maven
  - Vous devez mettre à jour le fichier pom.xml en suivant la documentation du plugin Maven Spring Boot
- Utilisez les 2 commandes Maven suivantes AVANT et après avoir mis à jour le fichier pom.xml
  - mvn help:effective-pom
  - mvn validate

# Exemple Maven : plugin Spring Boot 2/2

- On vous fournit le contenu du `pom.xml` (slide suivante) généré (et adapté) depuis **Spring Initializr**, avec les changements suivants :
  - Ajout d'un POM "parent" (dont on hérite) venant de Spring Boot
  - Simplification des properties (grâce au "POM parent")
  - Simplification des dépendances (pas de version à gérer, grâce au "POM parent")
  - Ajout d'une dépendance
  - Activation du plugin Spring Boot lors des phases de "build"

# Exemple Maven : pom.xml final

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cicdlectures</groupId>
  <artifactId>menuserver</artifactId>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.0</version>
  </parent>

  <properties>
    <java.version>11</java.version>
  </properties>

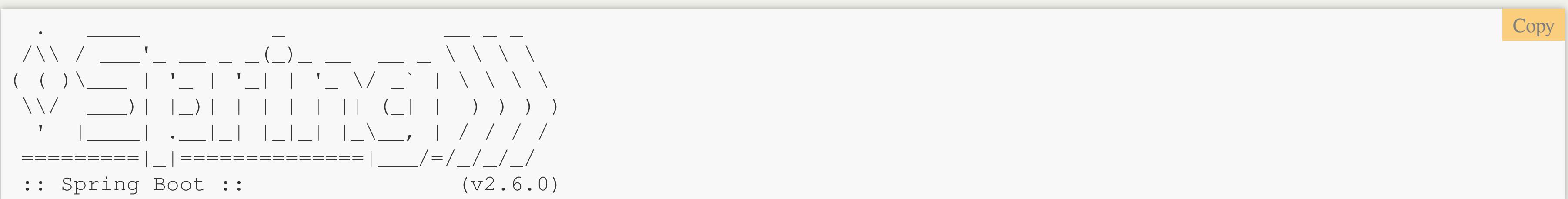
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
    </dependency>
  </dependencies>

```

Copy

# Exercice : Démarrer l'application

- **But:** Exécuter l'application à l'aide du plugin Spring Boot
- Sprint Boot fournit une phase Maven nommée `spring-boot:run` qui exécute l'application en mode "développement" sur le port 8080 local
  - Essayez d'appeler cette phase avec Maven
- Résultat attendu (une jolie bannière ASCIIArt):



# Exercice : Prévisualiser l'application

- Dans un second terminal de Gitpod, affichez la page web de l'application avec les commandes suivantes :
  - `gp url 8080` pour afficher l'URL publique de l'application correspondant au port 8080 local
  - `gp preview "$(gp url 8080) /"` pour prévisualiser le "endpoint" / dans un navigateur local
- La page d'accueil doit afficher HTTP/404
- Trouvez la page des menus qui doit répondre `[ ]` (liste vide en JSON)
  -  `src/*/controller/*.java`

# Checkpoint

- Spring Boot (et toutes ses dépendances) est configuré ✓
- mvn spring-boot:run compile et exécute l'application ✓
- Il faut encore fabriquer un fichier JAR  pour la production 🤔

⇒ reprenons avec Maven

# Exercice : Maven JAR Plugin

- **But:** Produire l'artefact JAR distribuable
- La génération du JAR est déclenchée lors de l'appel à mvn package :

```
ls -ltra ./target  
mvn clean # Nettoyez tout !  
ls -ltra ./target  
mvn package  
ls -ltra ./target # Est-ce que vous voyez un fichier JAR ?
```

Copy

- Exécution de l'application :

```
java -jar <chemin vers le fichier JAR>
```

Copy

- Même fonctionnement que précédemment (bannière, port 8080, endpoint /menus...)

# Exercice : Changer le nom de l'artefact final

- **But:** Produire un artefact JAR dont le nom est constant
- Quel est le nom de l'artefact généré ? Est-il constant ?
  - (SPOILER: ☺♀)
- En utilisant la documentation de référence <https://maven.apache.org/pom.html#the-basebuild-element-set>, adaptez votre pom.xml afin que le fichier généré se nomme **toujours** menu-server.jar.

# Solution : Changer le nom de l'artefact final

```
<build>
  <finalName>menu-server</finalName>
</build>
```

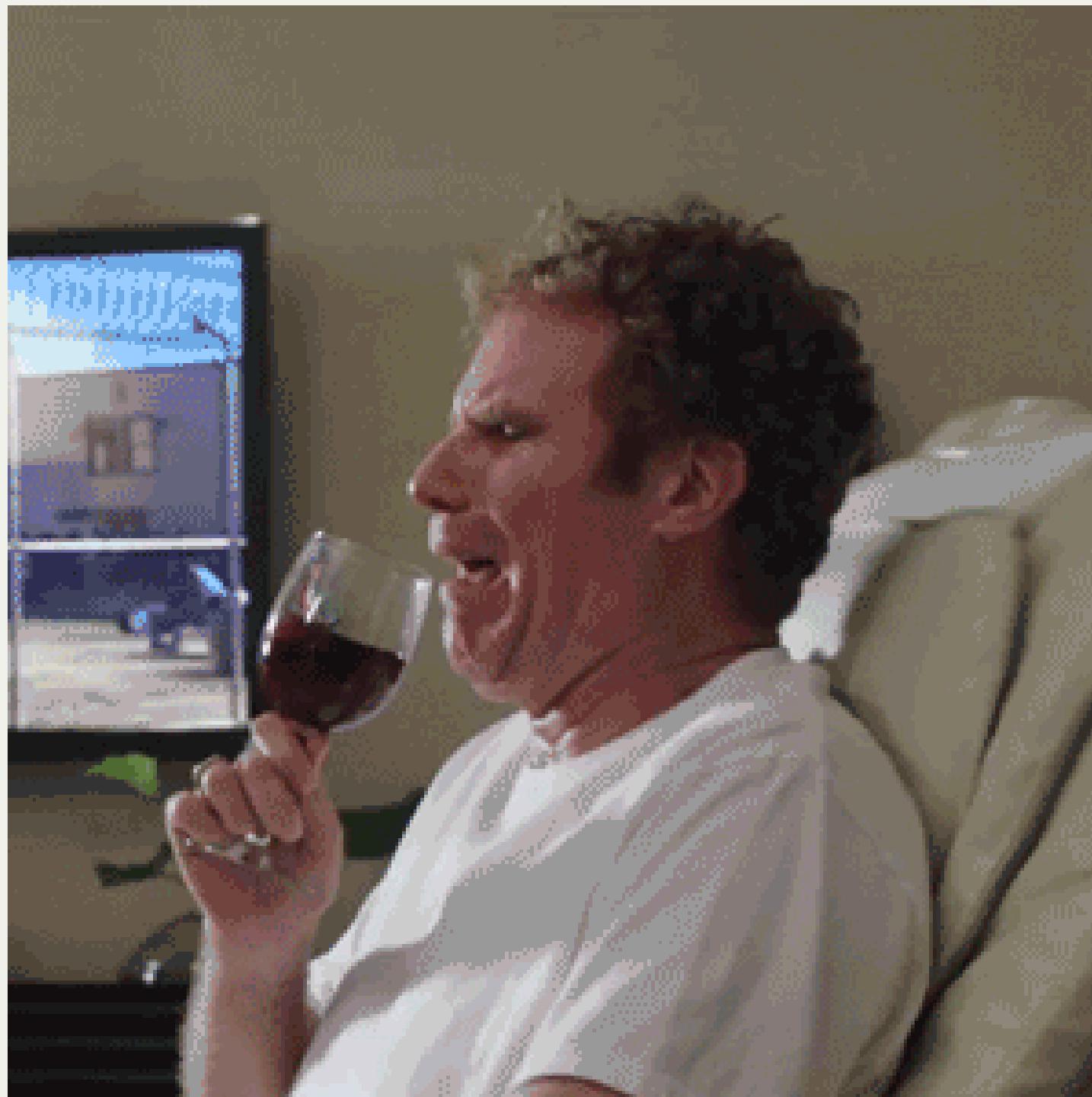
Copy

# 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.

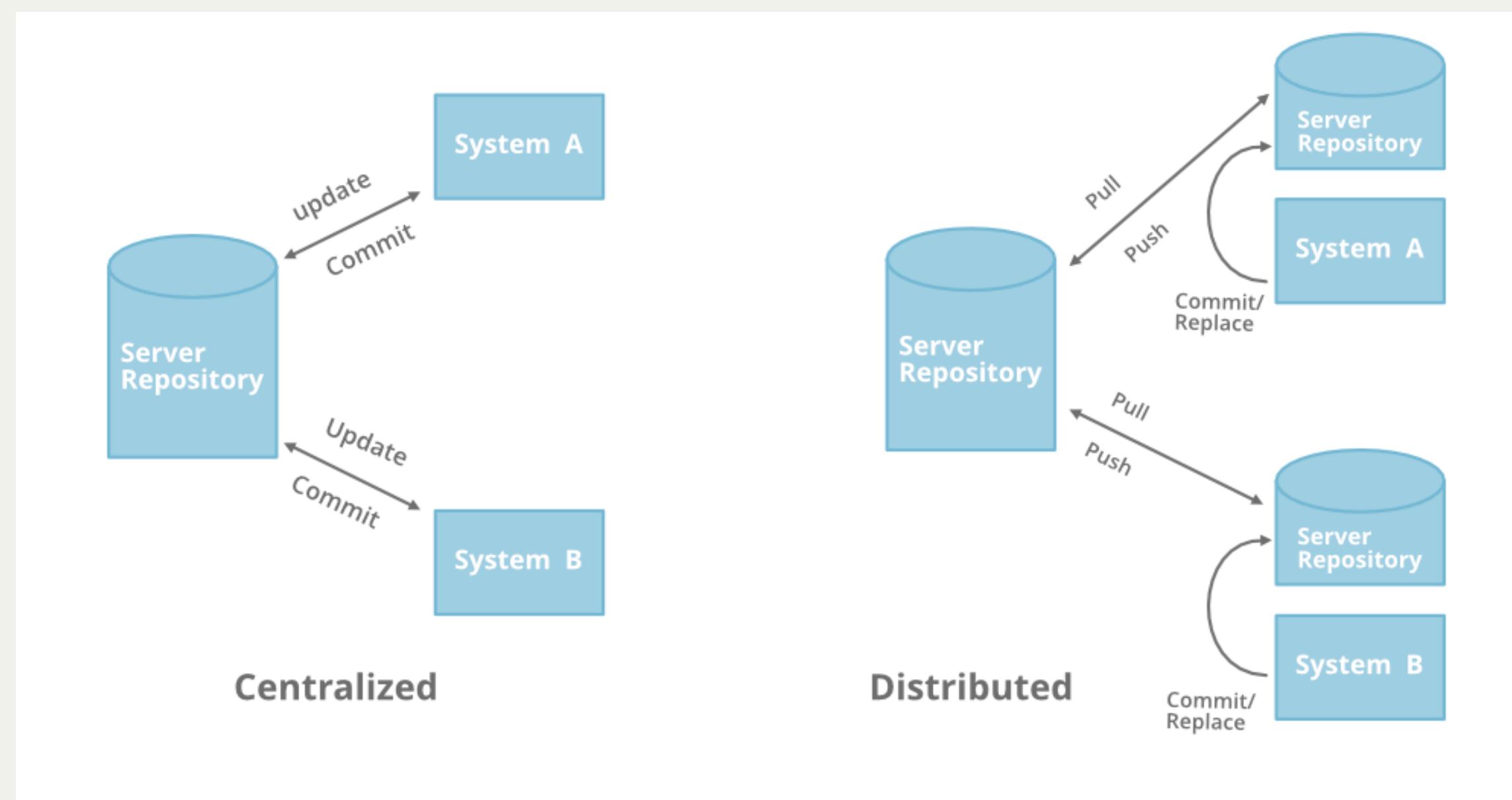
# Comment éviter ça ?

- 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**s.

# Centralisé vs Décentralisé



Source Geek for Geeks

Cela rends la manipulation un peu plus complexe, allons-y pas à pas :-)

# 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/...](https://github.com/)) de votre dépôt :-)

# 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

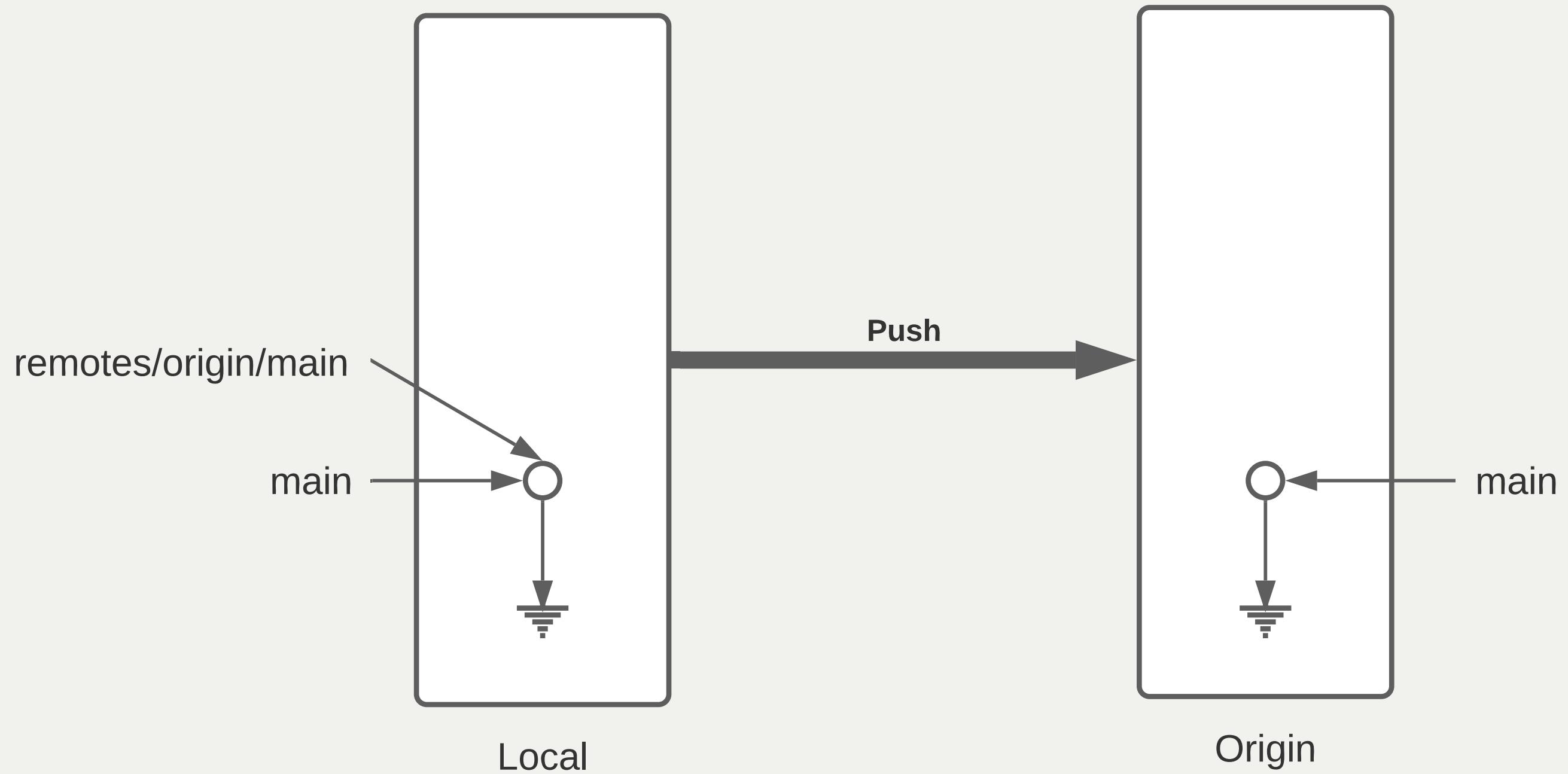
# Publier une branche dans sur dépôt distant

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 ?

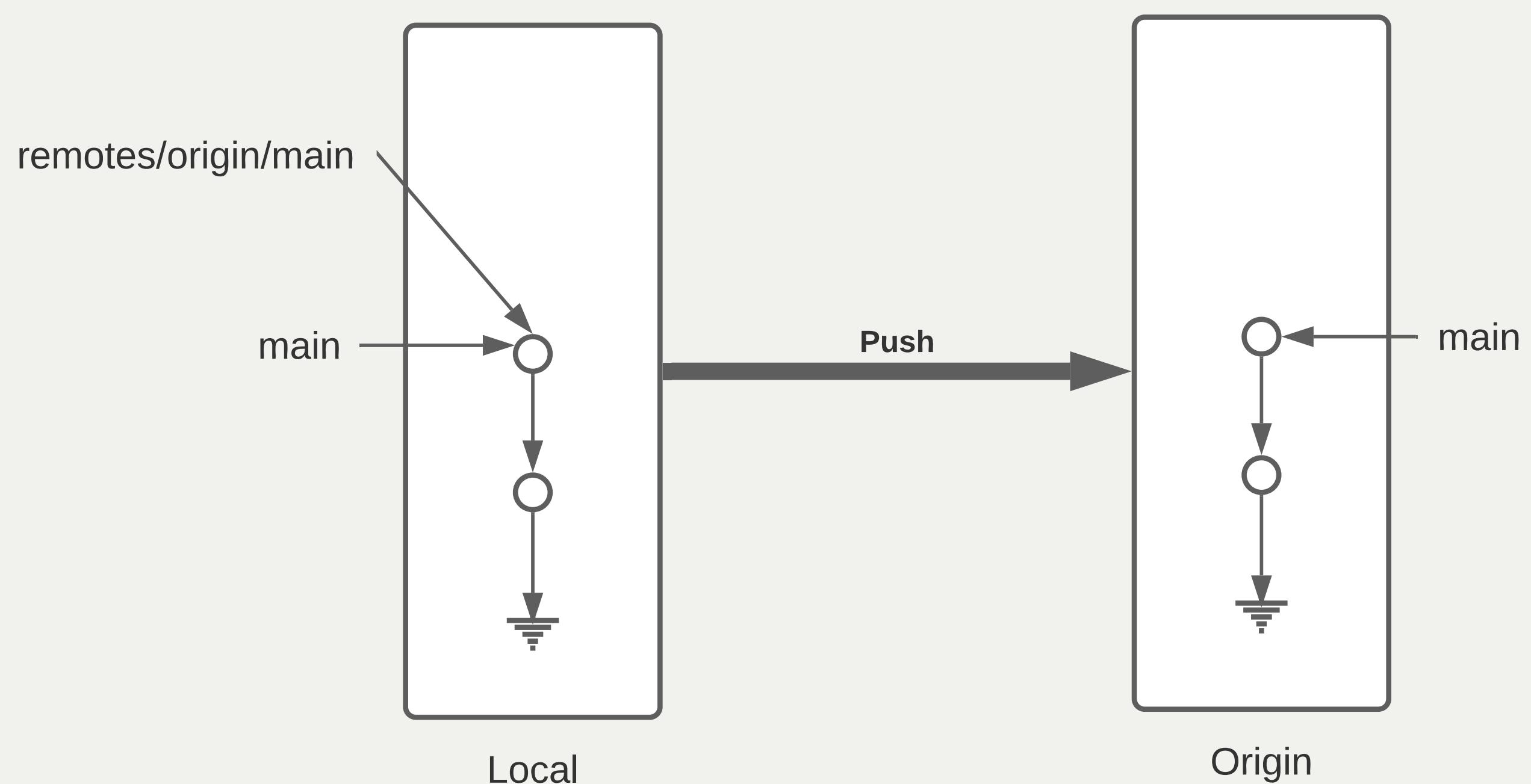


- git a envoyé la branche main sur le remote origin
- ... qui à accepté le changement et mis à jour sa propre branche main.
- git a créé localement une branche distante origin/main qui suis l'état de main sur le remote.
- Vous pouvez constater que la page github de votre dépôt affiche le code source

# 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>
```

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

Copy

```
# Notez que est listé remotes/origin/main
```

```
# Mets à 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

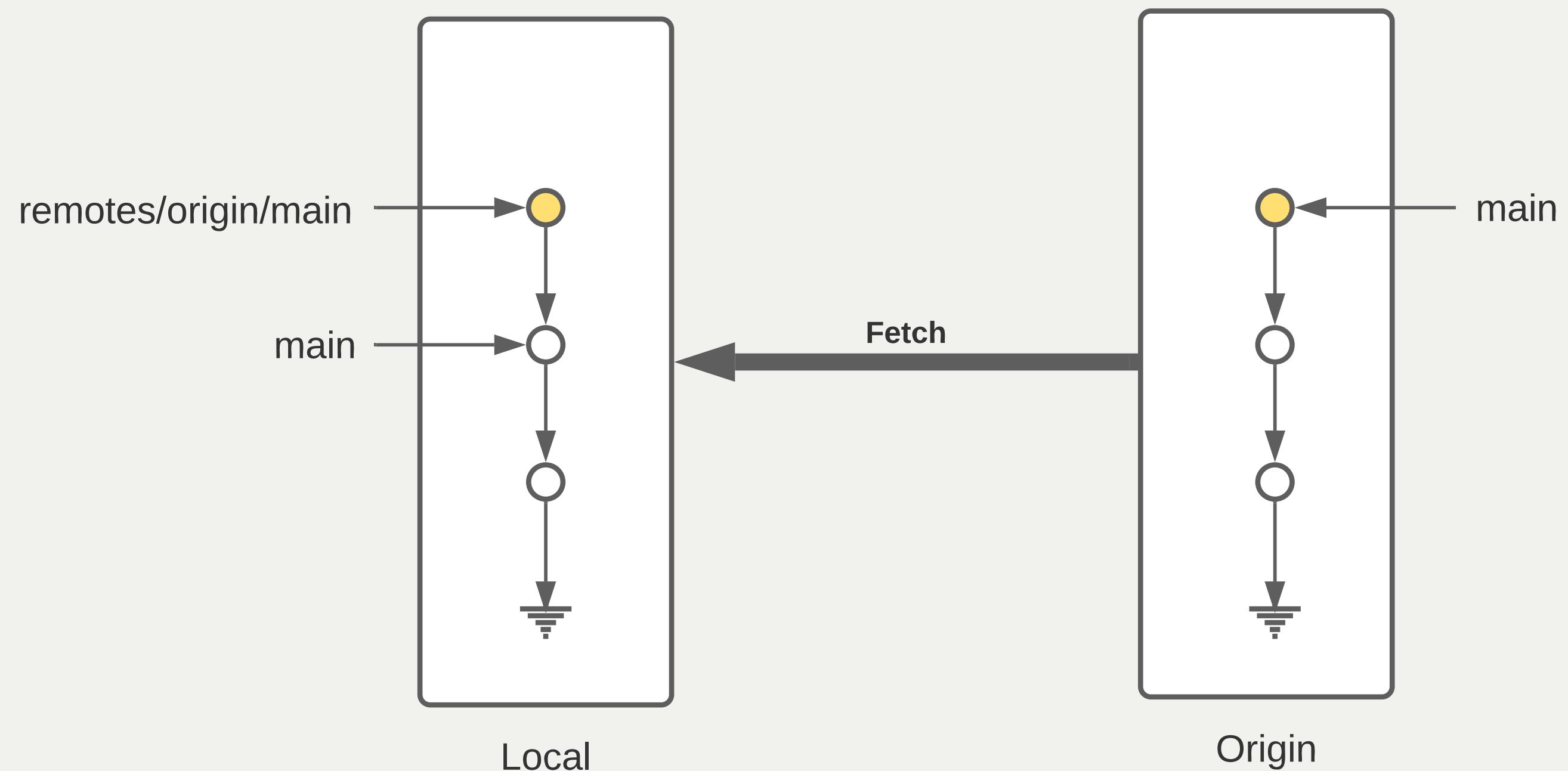
- Cliquez sur le bouton éditer en haut à droite du "README"
- Changez le contenu de votre README
- Dans la section "Commit changes"
  - Ajoutez un titre de commit et une description
  - Cochez "Commit directly to the main branch"
  - Validez

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
```

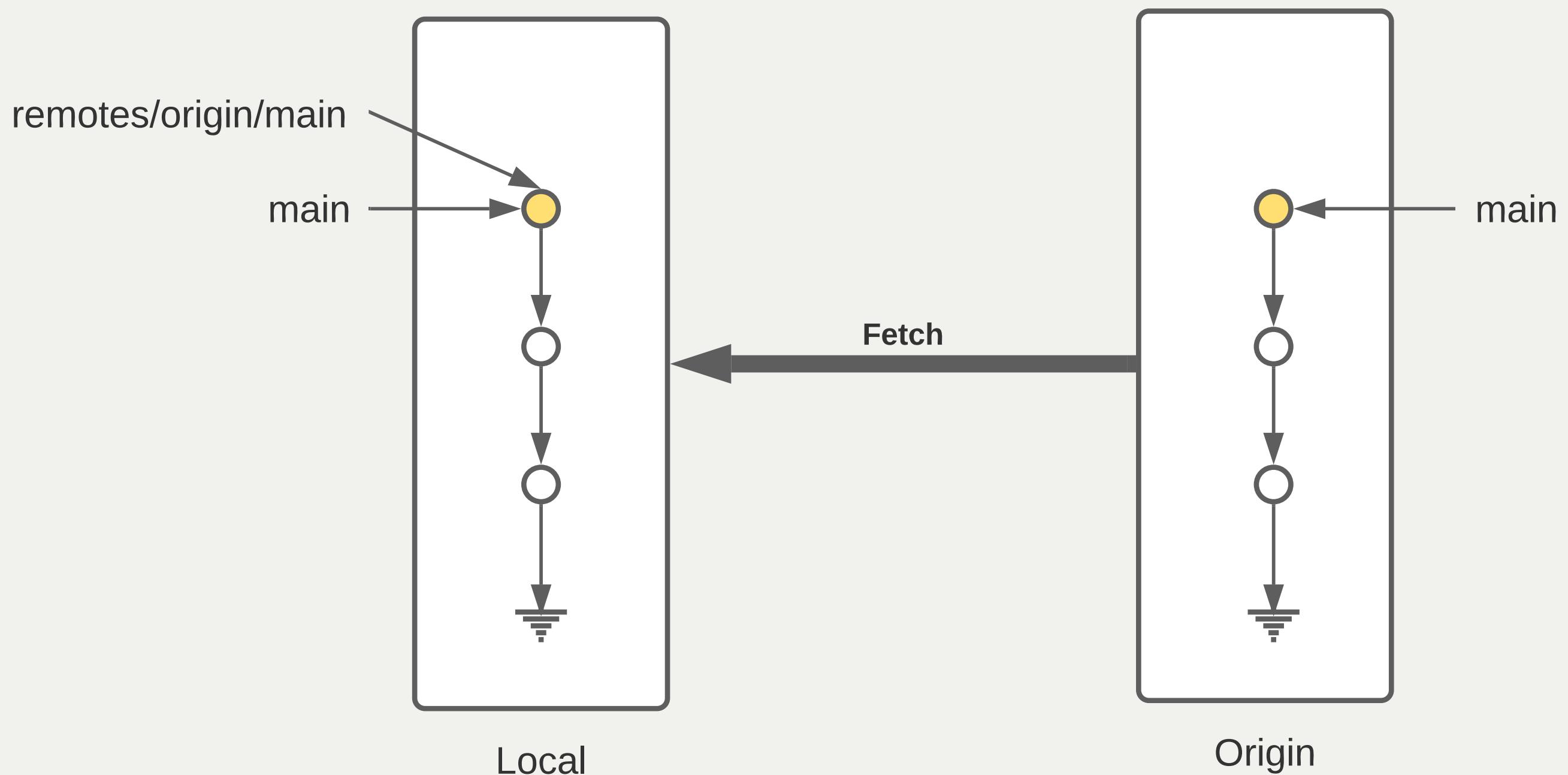
Copy

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
- Integration aux moteurs de CI/CD
- And so much more...

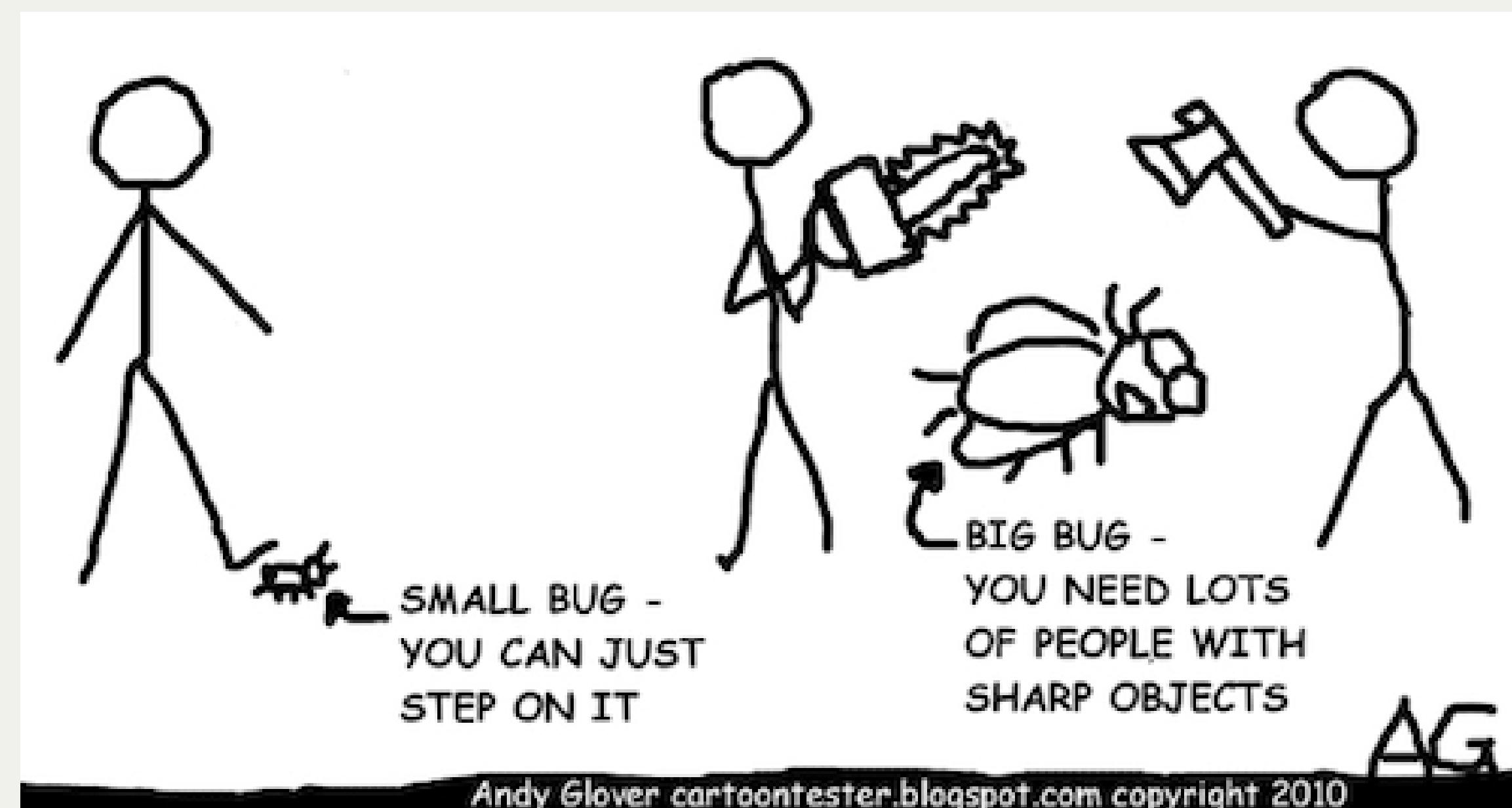
# Intégration Continue (CI)

*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



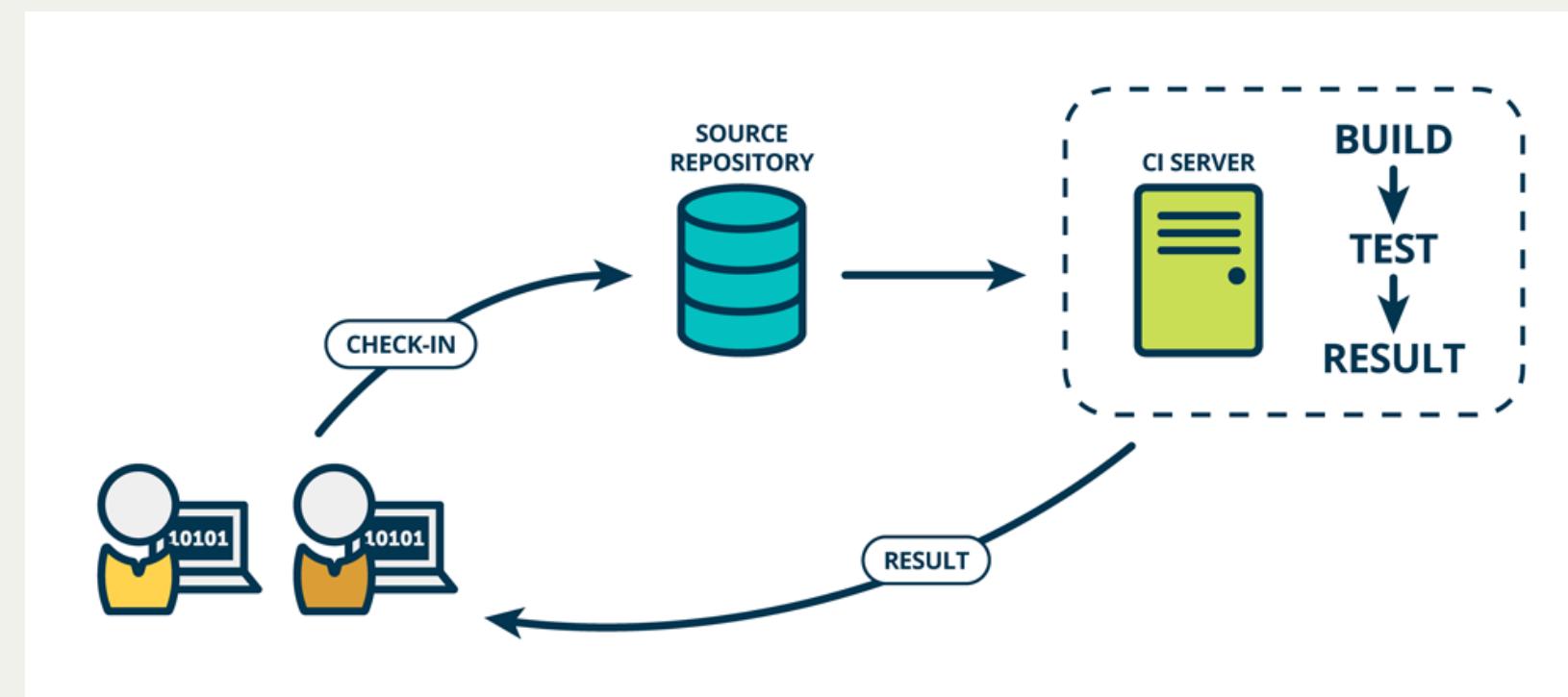
Source : <http://cartoontester.blogspot.be/2010/01/big-bugs.html>

# Qu'est ce que l'Intégration Continue ?

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



- 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

# Quelques moteurs de CI connus

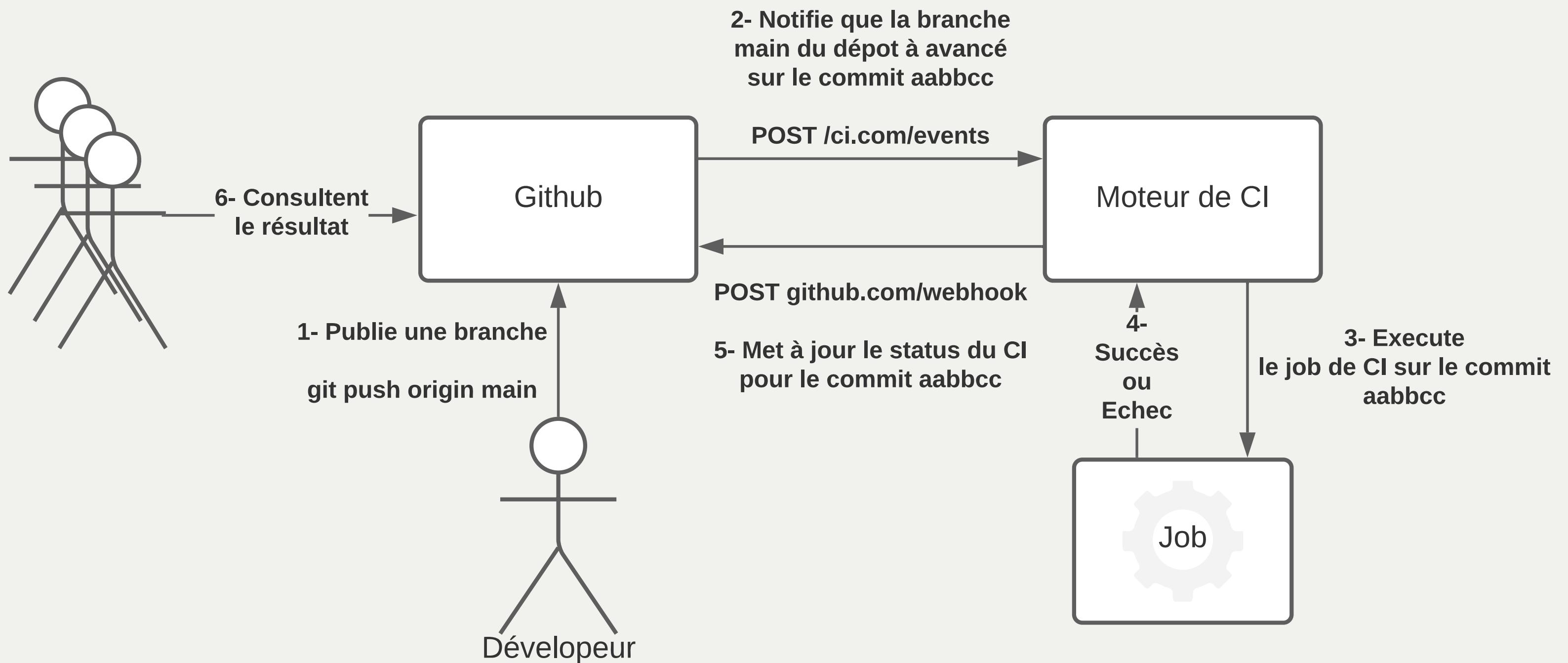
- A héberger soit-même : Jenkins, GitLab, Drone CI, CDS...
- Hébergés en ligne : Travis CI, Semaphore CI, Circle CI, Codefresh, GitHub Actions

# GitHub Actions

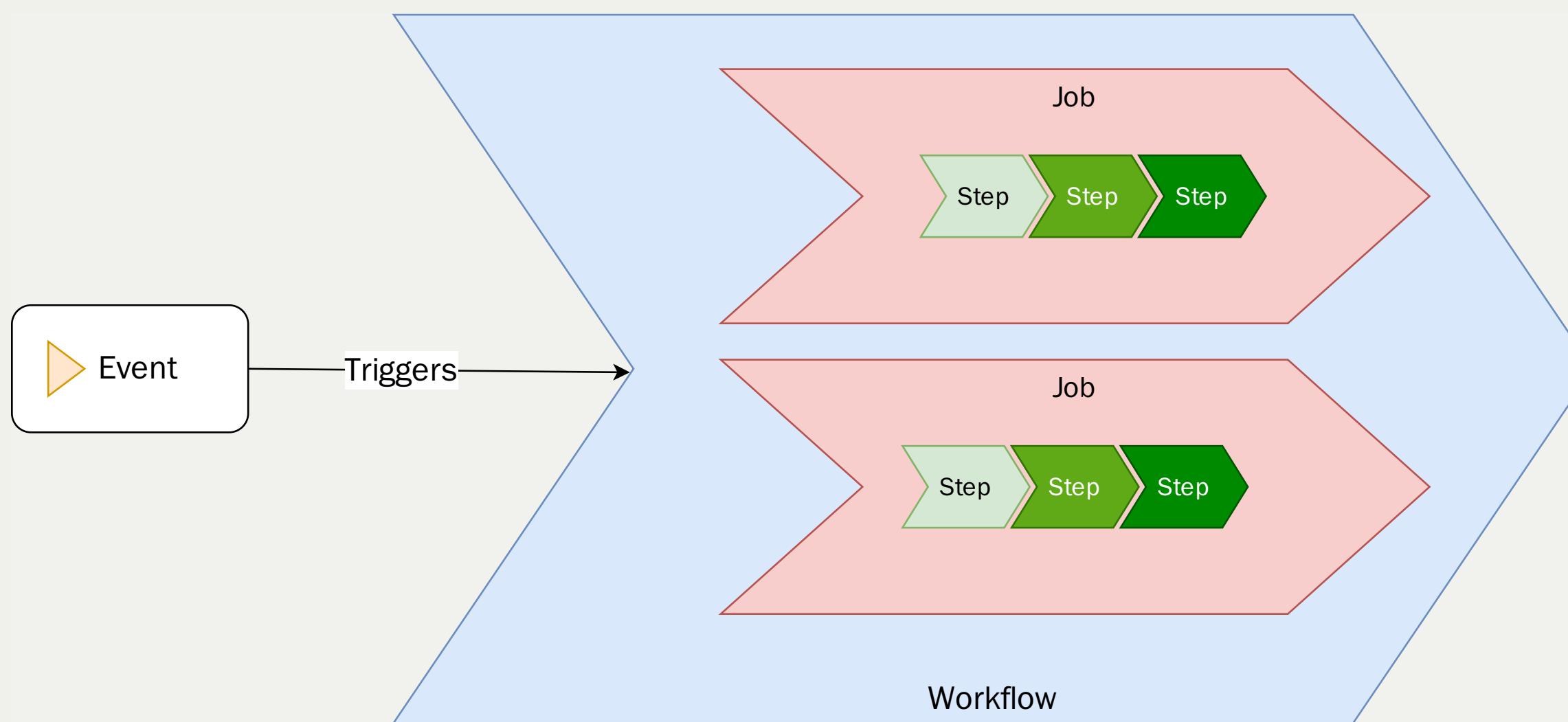
GitHub Actions est un moteur de CI/CD intégré à GitHub

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

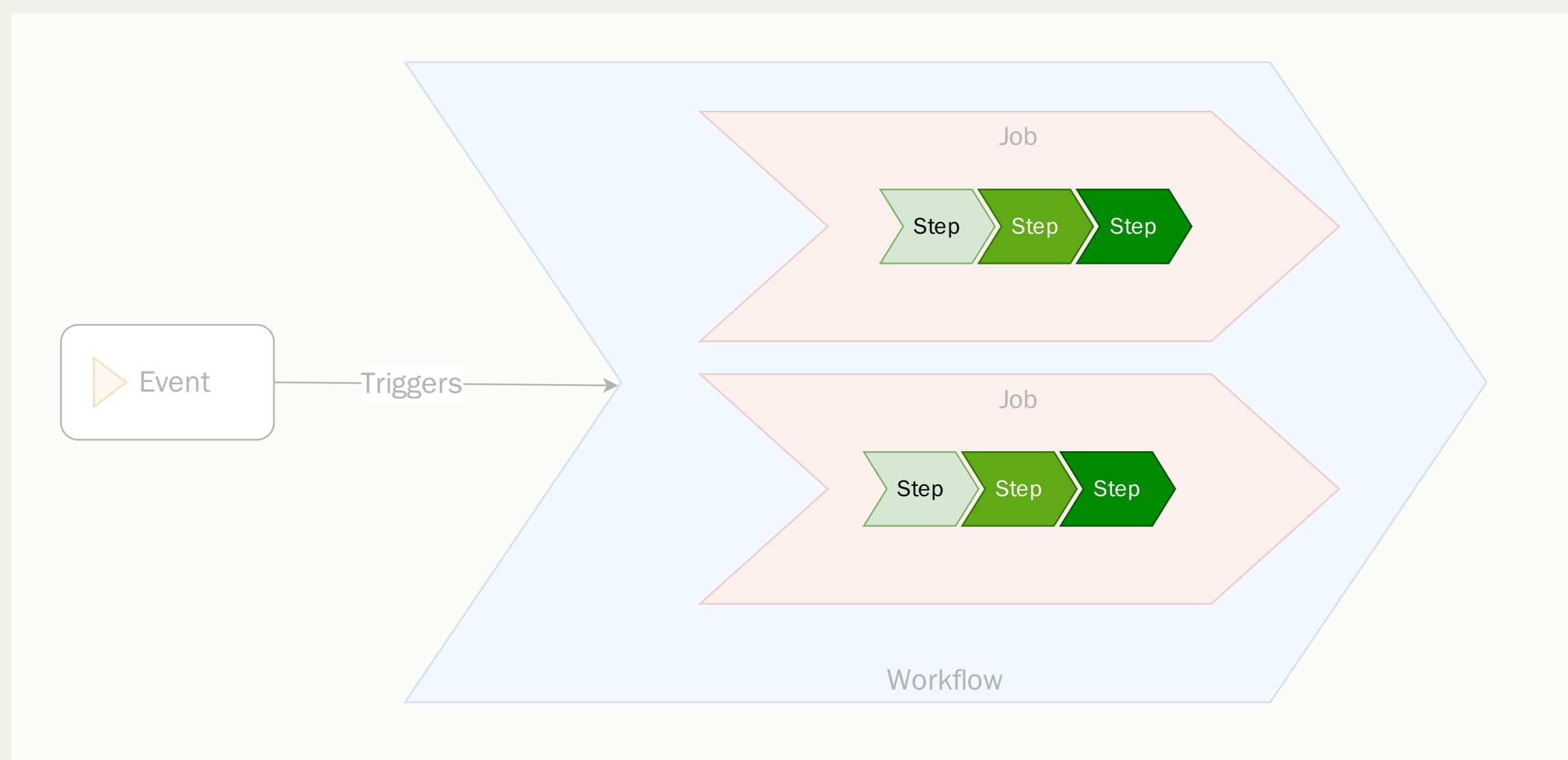
# Anatomie de déclenchement de GitHub Actions



# Concepts de GitHub Actions



# Concepts de GitHub Actions - Step 1/2



# Concepts de GitHub Actions - Step 2/2

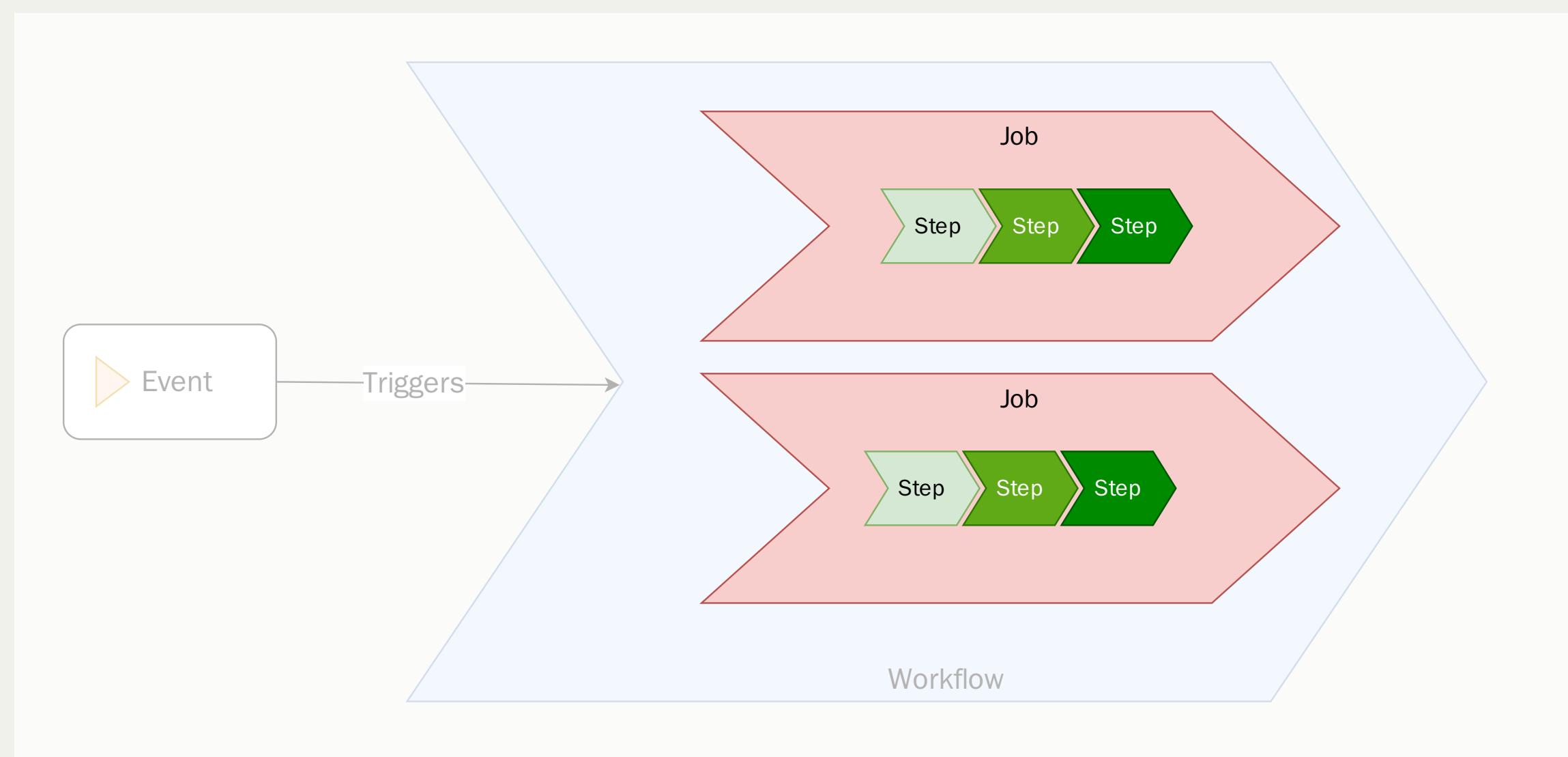
Une **Step** (étape) est une tâche individuelle à faire effectuer par le CI :

- 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

# 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-latest # cf. prochaine slide "Concepts de GitHub Actions – Runner"
    steps: # Collection de steps du job
      - name: 'Build the JAR'
        run: mvn package
      - name: 'Run Tests on the JAR file'
        run: mvn verify
  deploy: # 2nd job, identifié comme 'deploy'
    # ...
```

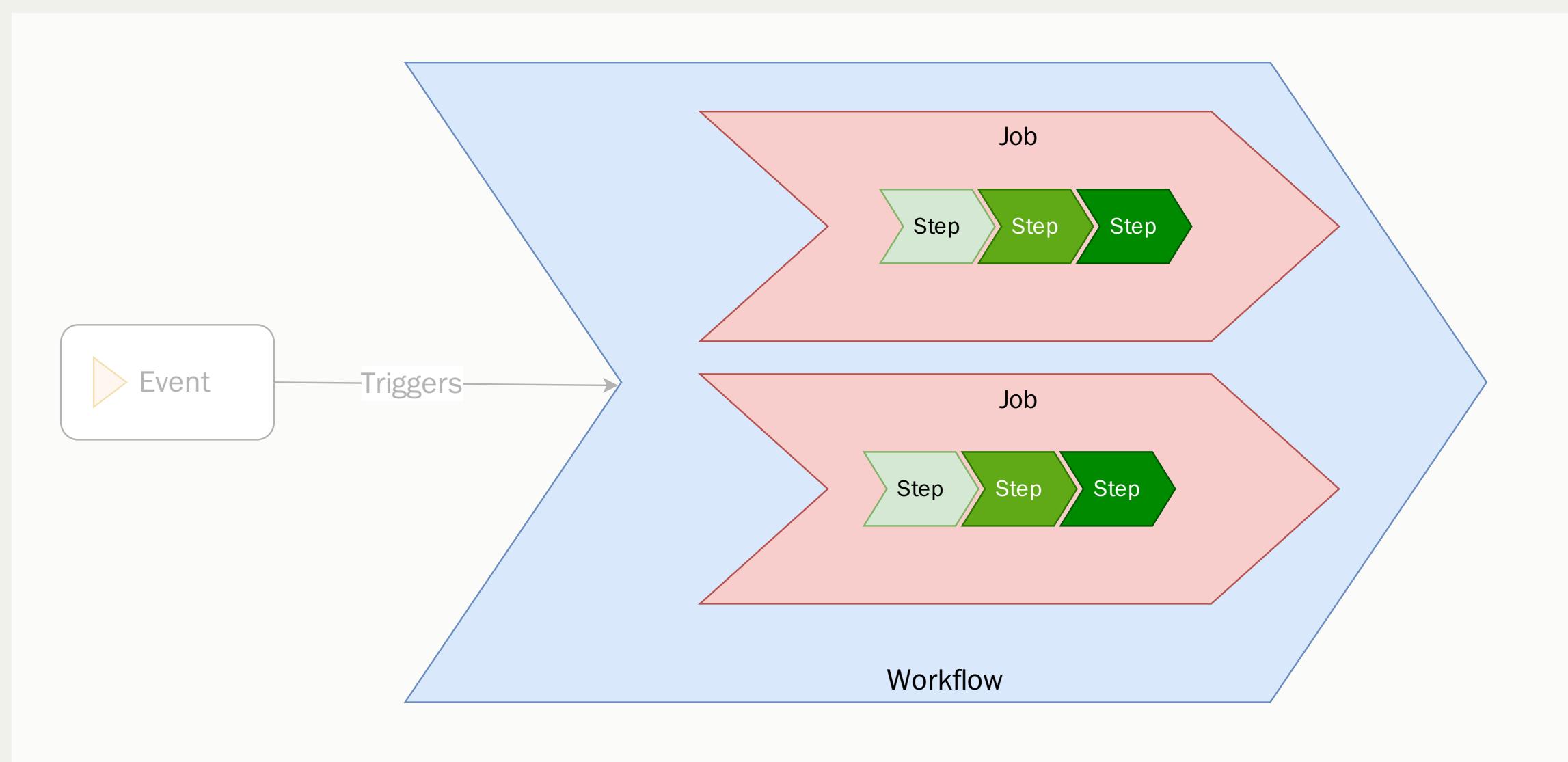
Copy

# Concepts de GitHub Actions - Runner

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

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

# Concepts de GitHub Actions - Workflow 1/2



# Concepts de GitHub Actions - Workflow 2/2

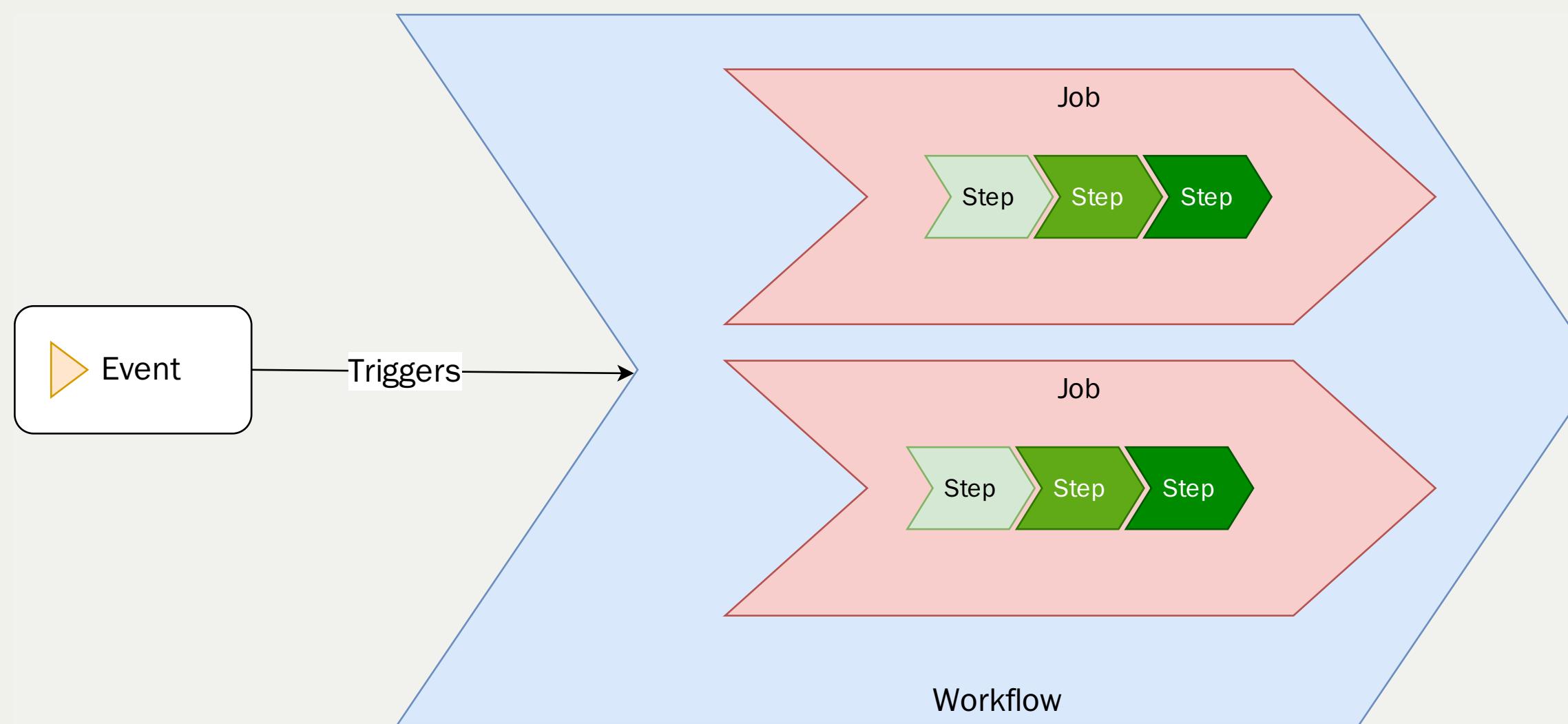
Un **Workflow** est une procédure automatisée composée de plusieurs jobs, décrite par un fichier YAML.

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

```
.github/workflows
├── ci-cd.yaml
├── bump-dependency.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-18.04
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm test
  test-mac:
    runs-on: mac-10.15
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm test
```

Copy

# Essayons GitHub Actions

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

# Exemple simple avec GitHub Actions

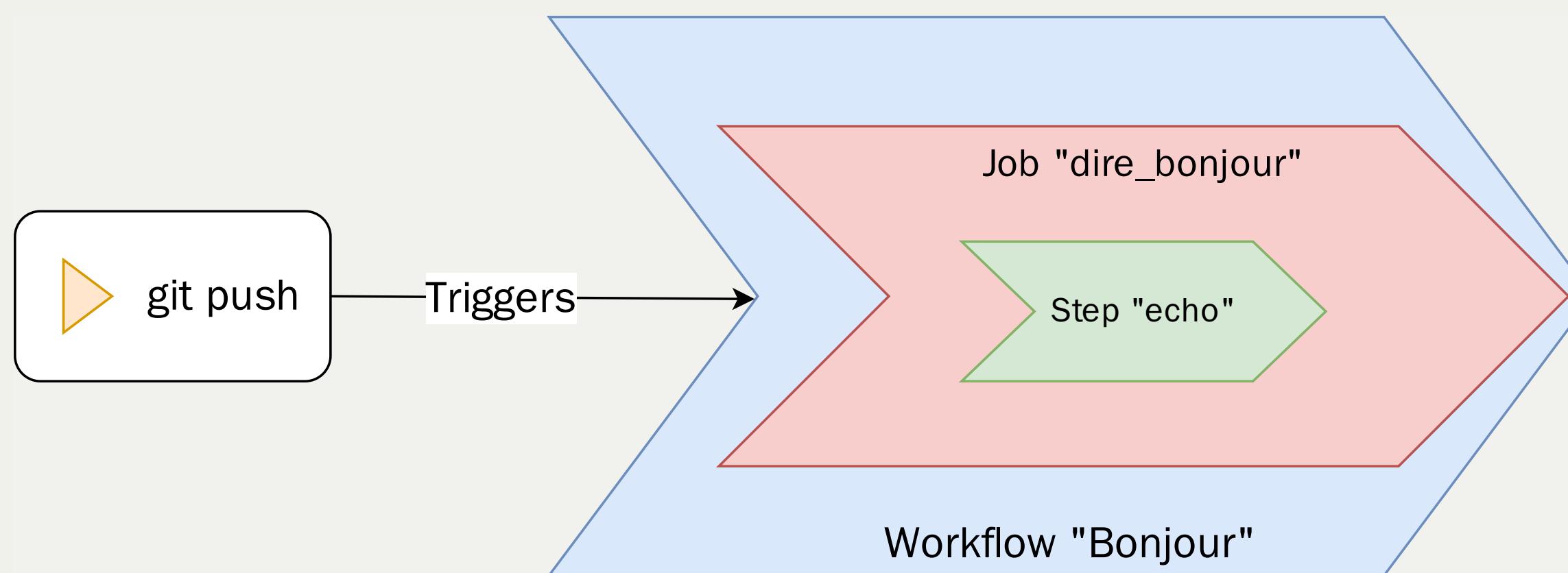
- Dans le projet "menu-server", sur la branch main,
  - Créez le fichier `.github/workflows/bonjour.yml` avec le contenu suivant :

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-20.04
    steps:
      - run: echo "Bonjour 🤘"
```

Copy

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

# Exemple simple avec GitHub Actions : 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-20.04
    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 “cat README.md” 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 "cat README.md" 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-20.04
    steps:
      - uses: actions/checkout@v2 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - 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

# 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 "cat README.md" 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: ( $\triangle$  sensible aux mises à jour,  $\checkmark$  facile à mettre en place)
  - Spécifier un environnement préfabriqué pour le workflow ( $\triangle$  complexe,  $\checkmark$  portable)
  - Utiliser les fonctionnalités de votre outil de CI ( $\triangle$  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...`)

# Solution : Personnalisation dans le workflow

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v2 # 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-20.04
    container:
      image: ghcr.io/cicd-lectures/gitpod:latest
      options: --user=root
    steps:
      - uses: actions/checkout@v2 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: cat README.md | cowsay
```

Copy

# Checkpoint



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



# Problème : Accélérer le workflow

- **Problème :**
  - Optimiser prématulement est contre-productif (commencez par faire un système qui marche comme prévu)
  - Mais il faut optimiser à un moment donné
- Essayons en ajoutant des étapes Maven en plus des étapes cowsay

# Exercice : Fonctionnalités du moteur de CI

- **But** : s'assurer que le workflow possède la même version de Java que GitPod...
  - ... le plus efficacement possible
- ⓘ C'est à vous de mettre à jour le workflow pour ajouter un 2nd job nommé `run_maven` dans le workflow :
  - On souhaite utiliser la même distribution majeure de Java JDK que dans GitPod (`java -version`, visez la même version mineure)
  - ⚒ Setup Java JDK (Verified Action)
  - Ce job doit récupérer le code puis afficher les versions de Maven, Java et le POM "effectif"

# Solution : Fonctionnalités du moteur de CI

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-20.04
    container:
      image: ghcr.io/cicd-lectures/gitpod:latest
      options: --user=root
    steps:
      - uses: actions/checkout@v2 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: cat README.md | cowsay
  run_maven:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-java@v2
        with:
          distribution: 'zulu'
          java-version: '11'
      - run: mvn -v
      - run: mvn help:effective-pom
```

Copy

# Orchestration des jobs

🤔 Exécution des 2 jobs : comment est-ce que Github a orchestré les 2 jobs ?

📝 Paralléliser les tâches non-dépendantes permet aussi d'obtenir des retours au plus tôt

# Exercice : Intégration Continue du projet "menu-server"

⚠️ ♀ C'est à vous de modifier le projet "menu-server" pour :

- Supprimer le workflow `bonjour.yaml`
- N'avoir plus qu'un seul job nommé `build` dans un workflow nommé "CI" (fichier `ci.yaml`) qui va :
  - Récupérer le code
  - Installer la même version de Java (mineure) que dans GitPod
  - Générer l'artefact "JAR" de l'application à l'aide de Maven
  - Uploader l'artefact "JAR" dans GitHub Actions pour le conserver
    -  (`actions/upload-artifact`)

# 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 ?

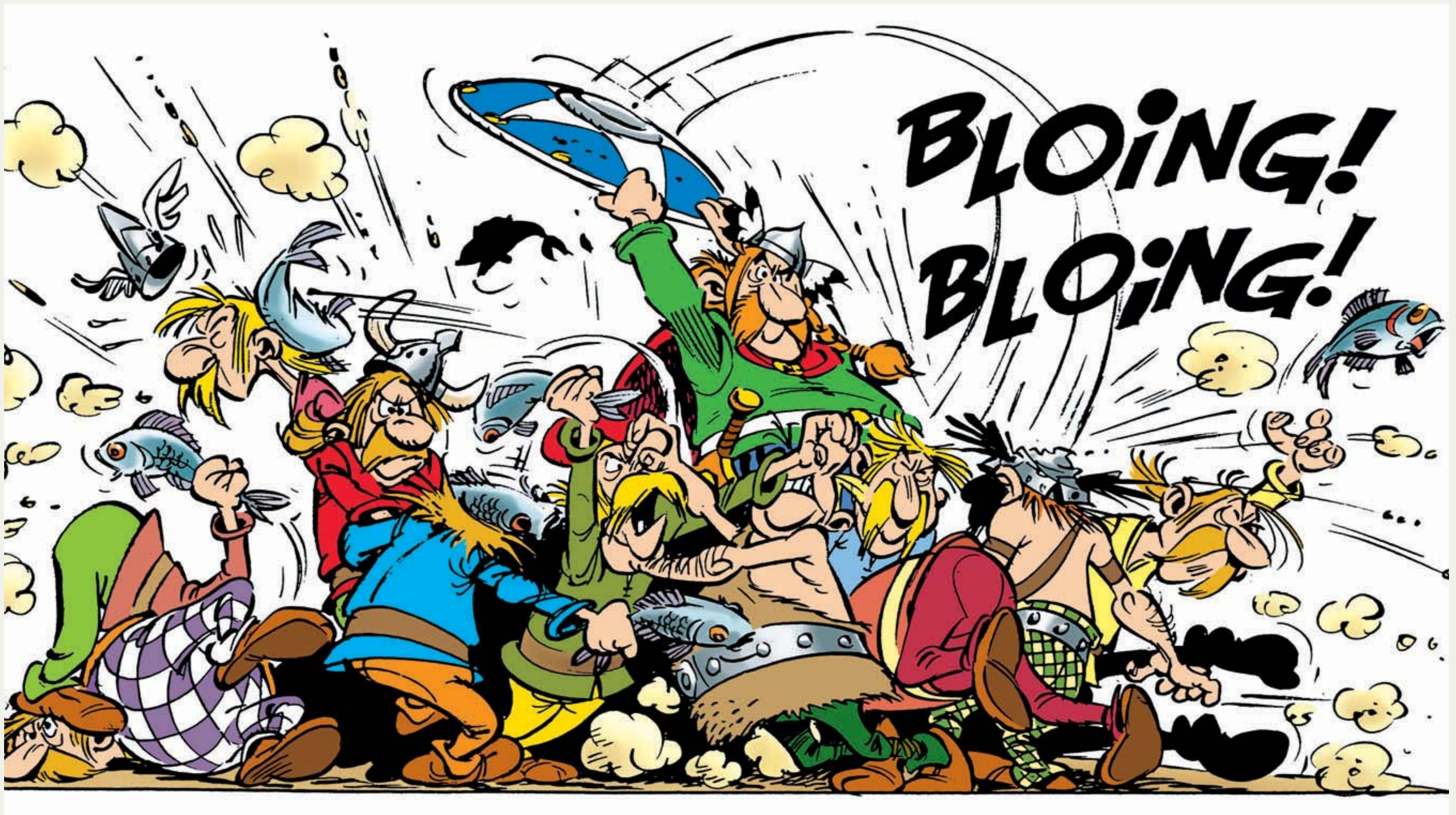
**Collaborer c'est pas évident, mais il existe des outils et des méthodes pour vous aider.**

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 !

... et (souvent) ça finit comme ça !

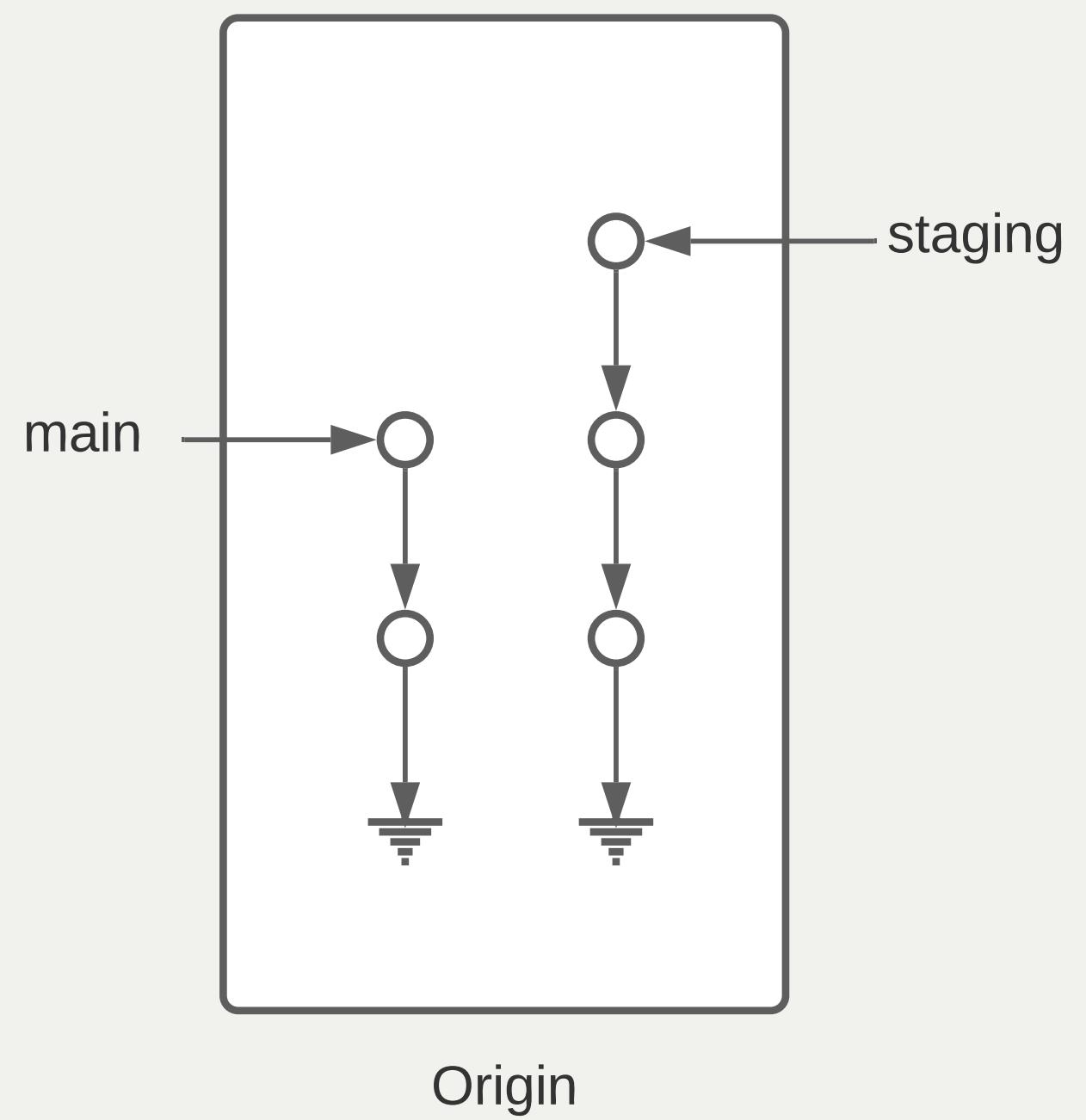


# Un Example de Git Flow

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

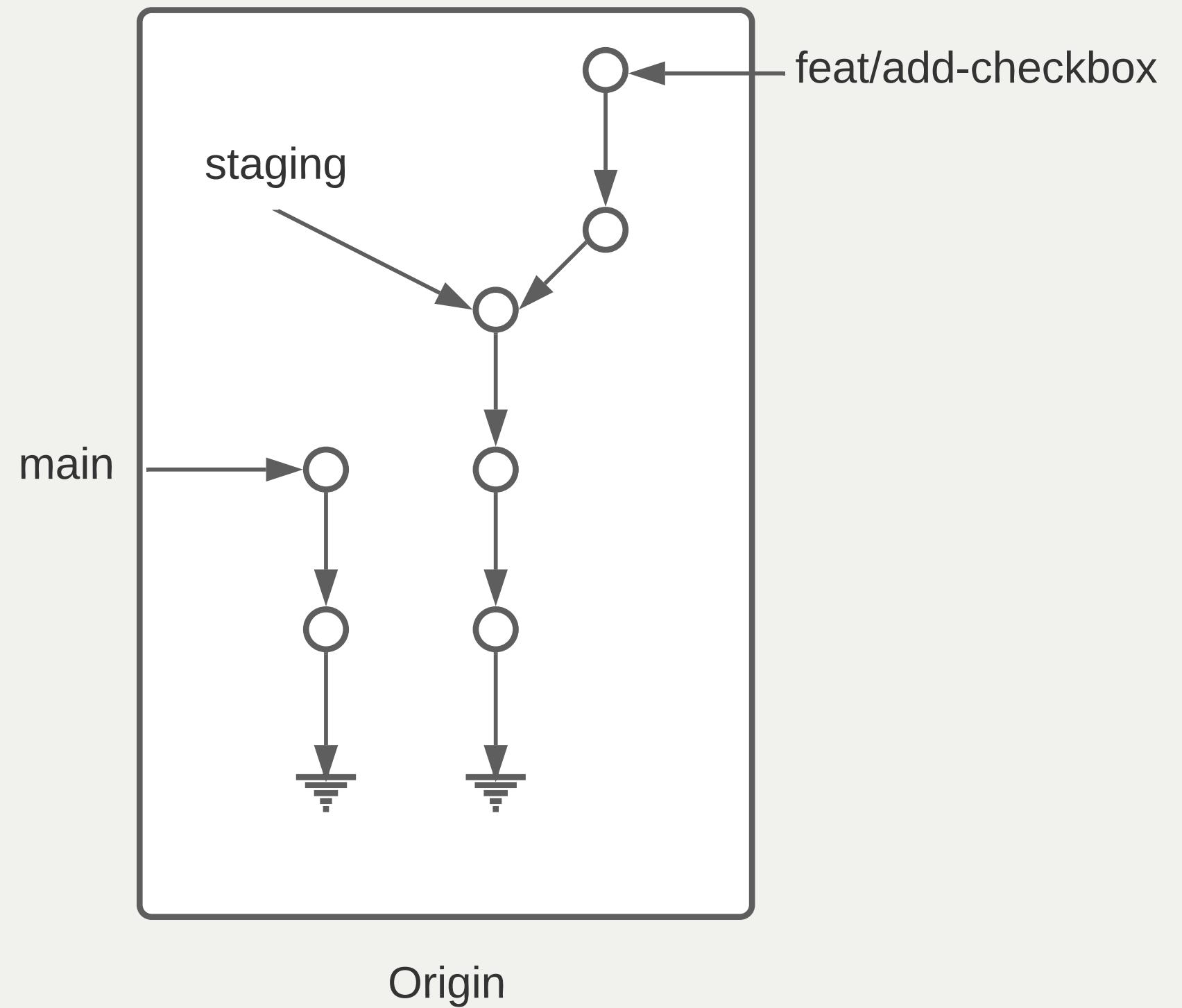
# Gestion des branches

- 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

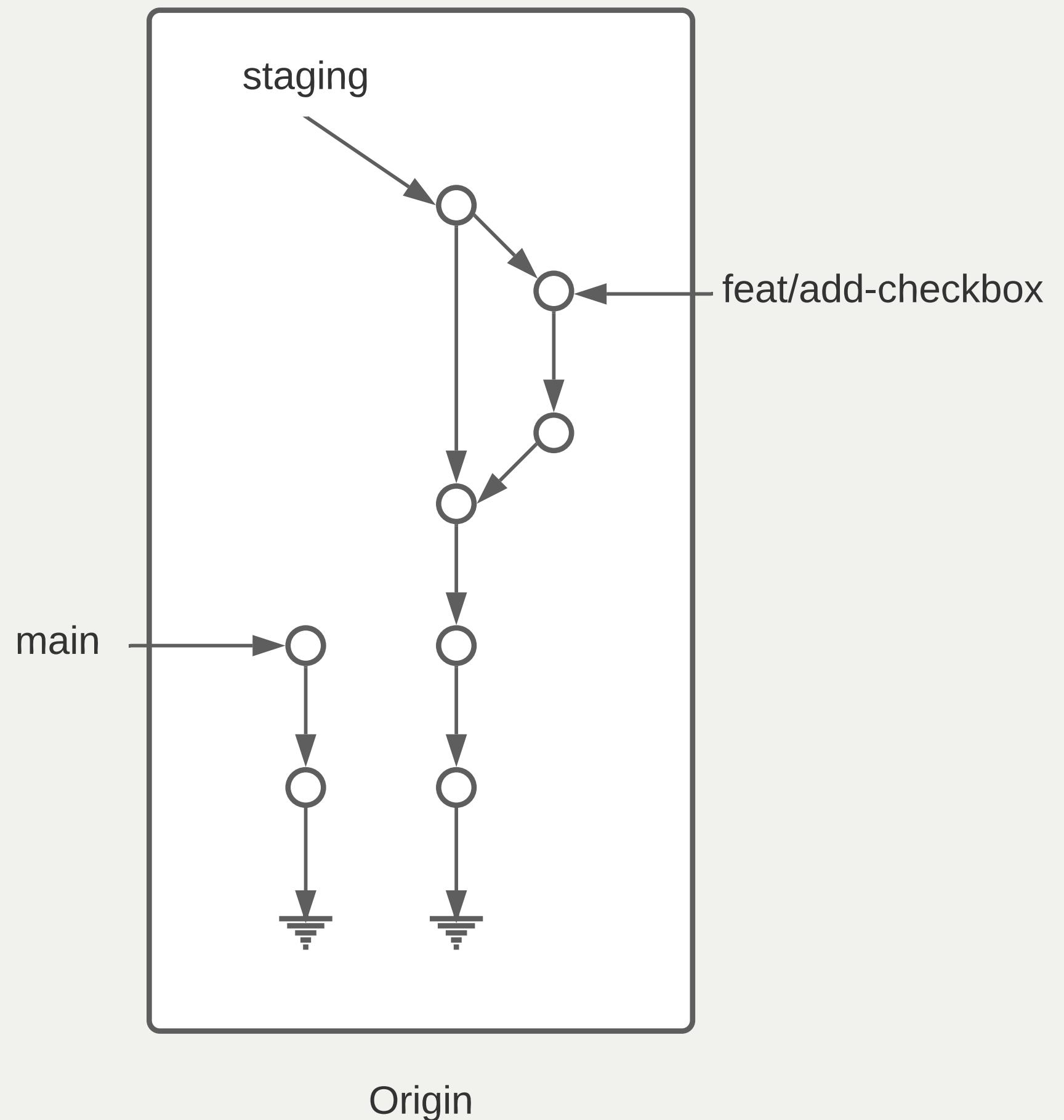


# Gestion des branches

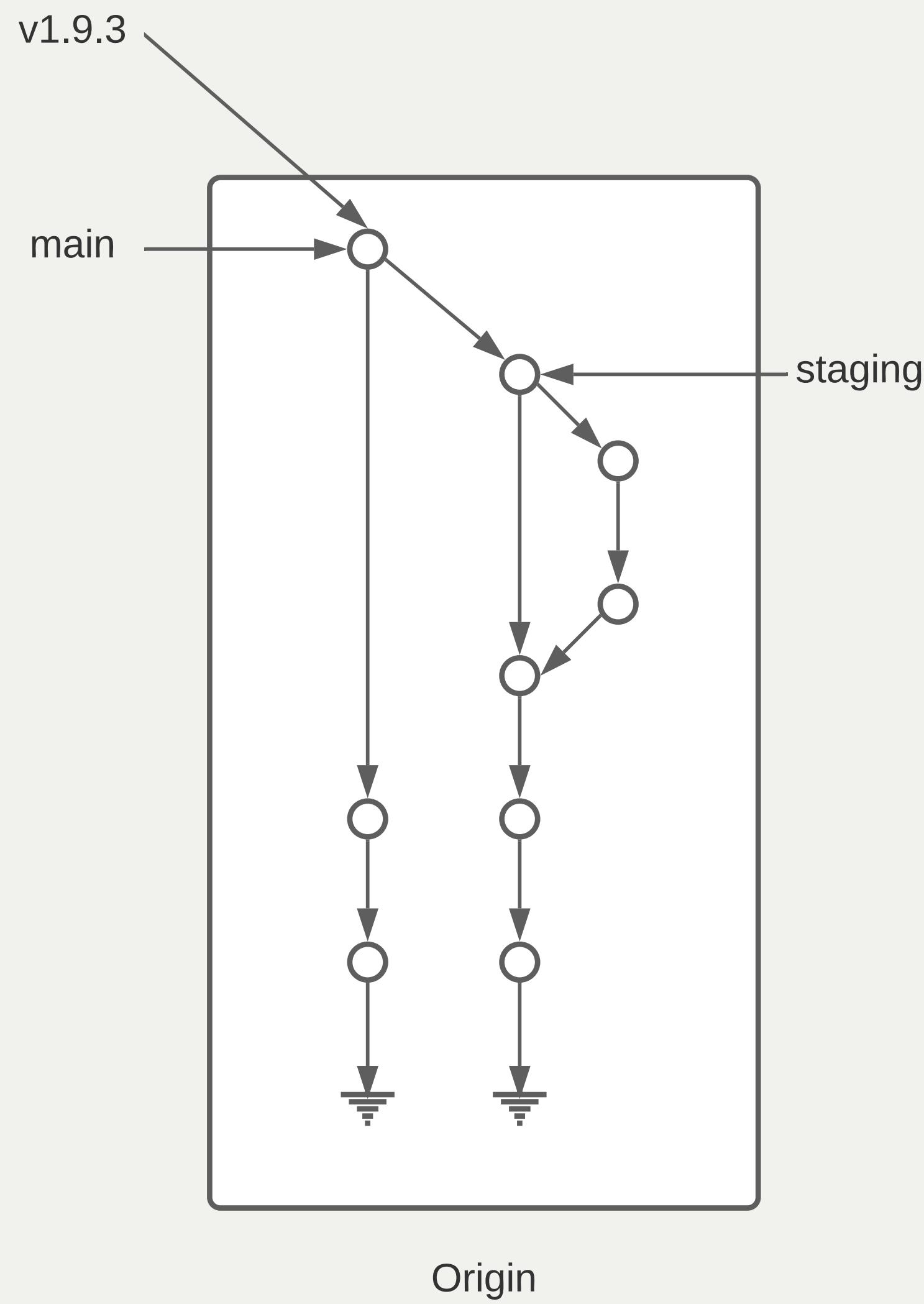
- Chaque groupe de travail (développeur, binôme...)
  - Crée une branche de travail à partir de la branche staging
  - Une branche de travail correspond à **une chose à la fois**
  - Pousse des commits dessus qui implémentent le changement



Origin



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



# 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!

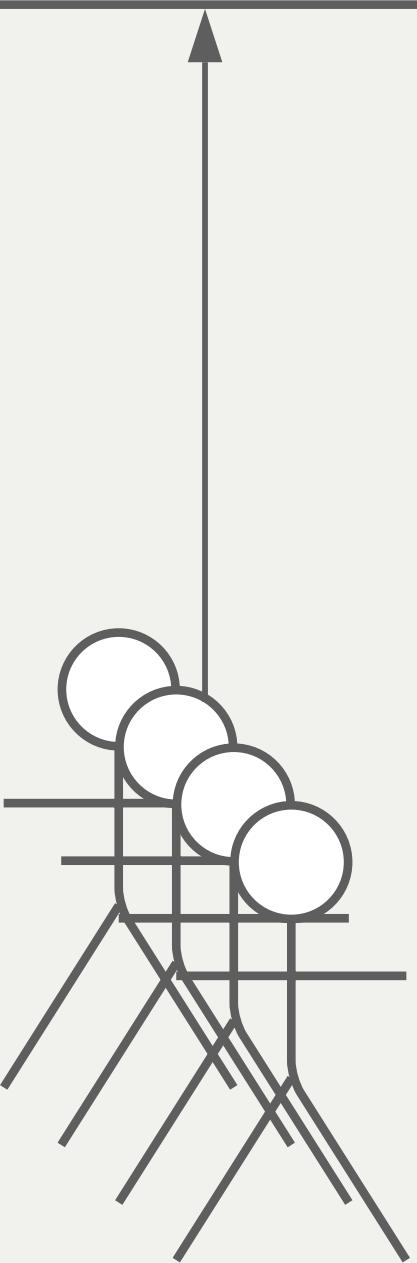
# Un remote pour les gouverner tous

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

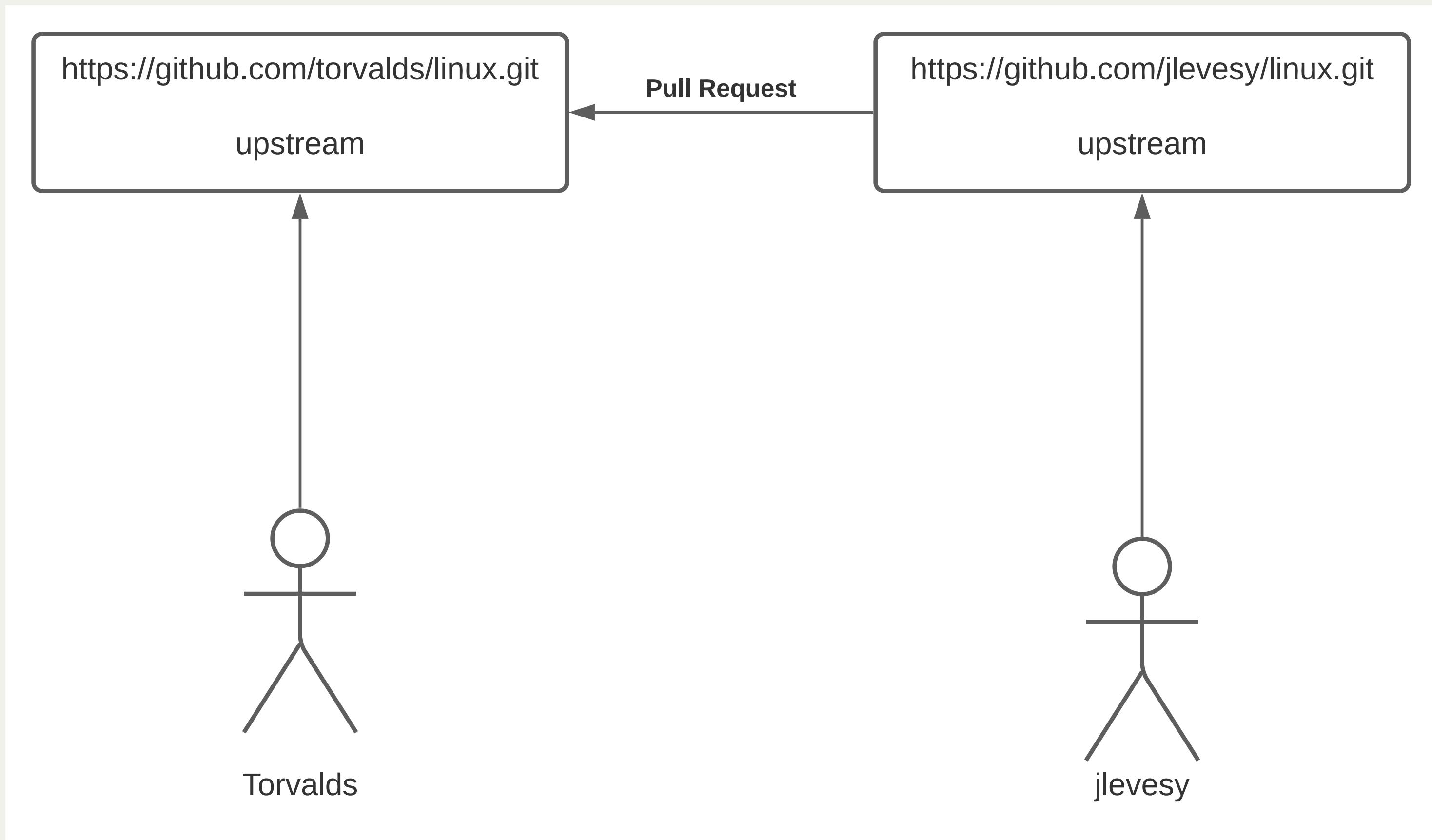
<https://github.com/torvalds/linux.git>

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 !



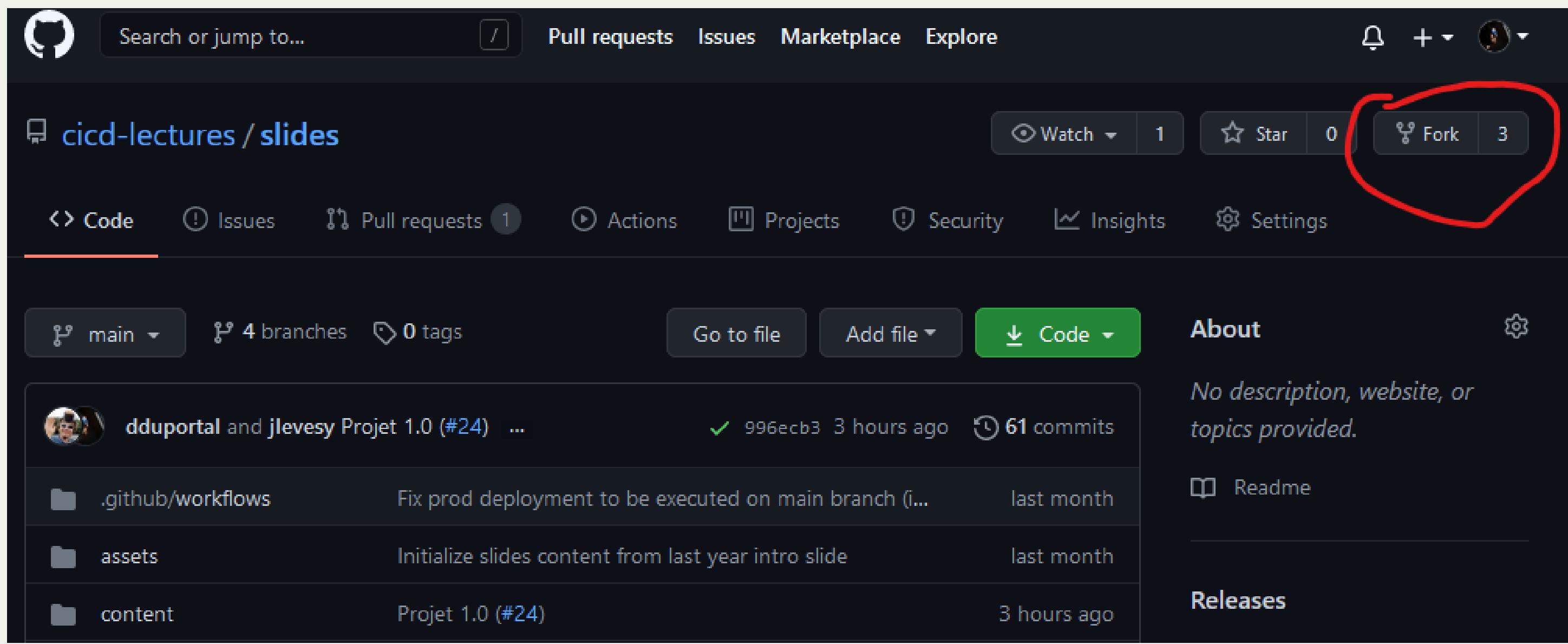
# Forks ! Forks everywhere !

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)

# Exercice: Créez un fork

- Nous allons vous faire forker vos dépôts respectifs
- Trouvez vous un binôme dans le groupe.
- Rendez vous sur cette page pour enregistrer votre binôme, et indiquez les liens de vos dépôts respectifs.
- 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 checkout -b implement-delete
```

Copy

# Exercice: Contribuez au projet de votre 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 pas.

Voici un petit article bien utile pour vous aider!

# 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

# Et on vérifie que le menu est bien supprimé
curl localhost:8080/menus
```

# 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.

# Exercice: Contribuez au projet de votre 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

## **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

(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.

# Ajout des Outils de Tests Automatisés au Projet (1/3)

L'execution de tests nécessite un outillage non ajouté au projet

- Framework d'écriture et d'execution de tests: JUnit
- Librairie de création de mocks: Mockito
- Plugin maven de lancement de tests: surefire

# Ajout des Outils de Tests Automatisé au Projet (2/3)

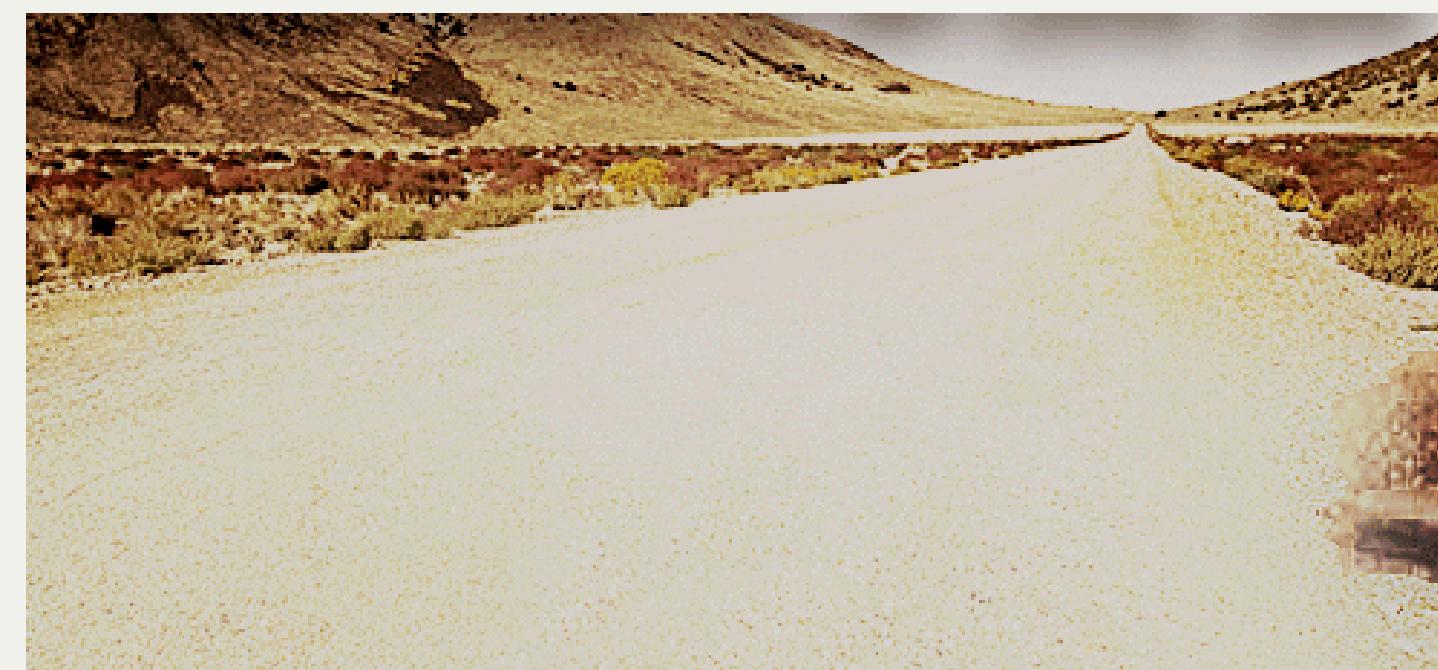
Ajoutez le bloc suivant au pom.xml

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
```

Copy

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

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



# 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.

# Exercice : Corriger un Bug (9/11)

```
@Test
@DisplayName("lists all known menus")
public void listsKnownMenus() {
    // Défini une liste de menus avec un menu.
    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 définit wantMenus, les résultats attendus
Iterable<MenuDto> wantMenus = Arrays.asList(
    new MenuDto(
        Long.valueOf(1),
        "Christmas menu",
        new HashSet<>(
            Arrays.asList(

```

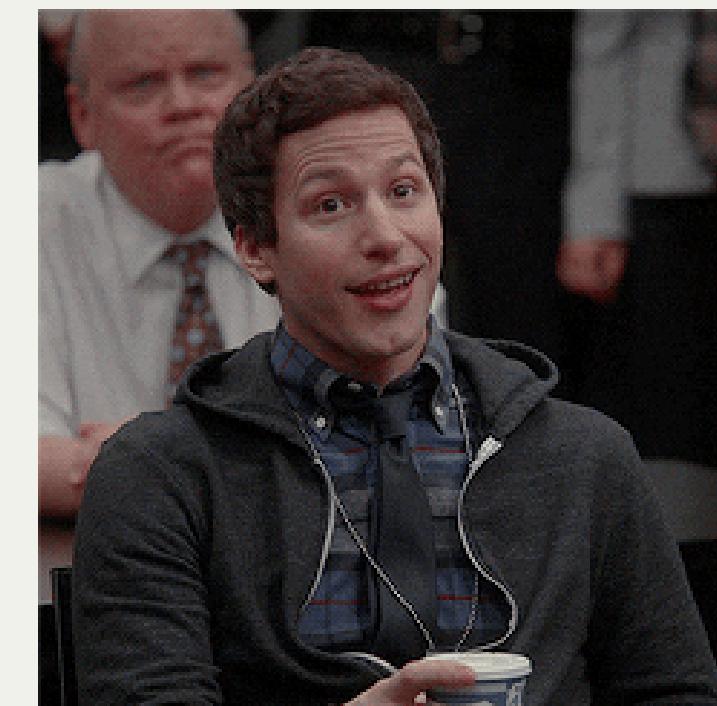
Copy

# 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:

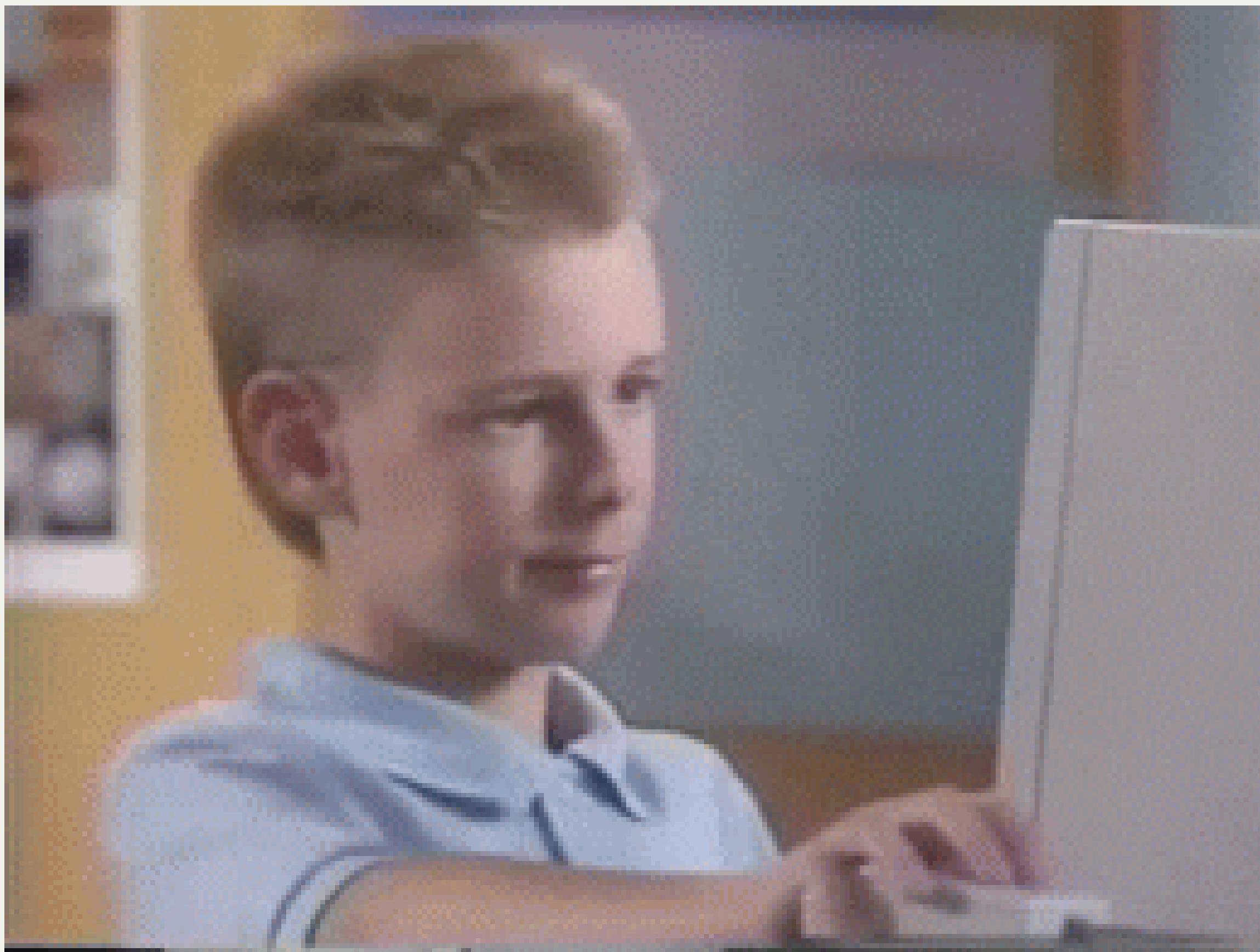
```
[ERROR] Failures:
[ERROR]   ListMenuServiceTests.listsKnownMenus:66
expected:
<[MenuDto(id=1, name=Christmas menu, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey) ]) ]>
but was:
<[MenuDto(id=1, name=TEST TODO, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey) ])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

Copy



# Exercice : Corriger un Bug (11/11)

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



# Tester la classe CreateMenuService

- Le CreateMenuService implémente la logique de création de menu au sein de l'application.
- Elle enregistre en base un nouveau menu avec tous ses plats
- Et répond le nouveau menu enregistré en base
- En revanche, elle implémente une logique de déduplication des plats par nom:
  - Si un plat portant le même nom existe déjà en base, il est réutilisé (pour éviter la duplication)

On doit faire en sorte de vérifier ce comportement.

# Vérifier les interactions avec les classes Mockées (1/2)

- Le but d'un test unitaire est de valider le comportement d'une méthode
- Par comportement nous entendons:
  - Retourner les bonnes valeurs
  - Appelle les bonnes méthodes des classes dont elle dépend, en passant les bons paramètres.

# Vérifier les interactions avec les classes Mockées (2/2)

```
// configure le mock pour qu'il retourne une instance de menu
when(menuRepository.save(any(Menu.class))).thenReturn(storedMenu);

// On appelle le code à tester...

// On déclare un ArgumentCaptor<Menu> (qui sert à capturer un argument)
ArgumentCaptor<Menu> savedMenuCaptor = ArgumentCaptor.forClass(Menu.class);

// On vérifie que la méthode `save` du menu repository à été appelée une seule fois
// et on capture l'argument avec lequel elle a été appelée (le menu).
verify(menuRepository, times(1)).save(savedMenuCaptor.capture());

// On récupère la valeur capturée pour pouvoir faire des assertions dessus.
savedMenu = savedMenuCaptor.getValue()
```

Copy

# Exercice: Écrire un test qui prouve que CreateMenuService sauvegarde le menu

## Plan de test

- On crée une instance DishDTO qui représente le menu à créer
- On crée une instance de Menu qui représente la valeur répondue par la base de données
- On appelle CreateMenuService.createMenu avec notre DTO
- On capture le menu enregistré et on vérifie qu'il a les bonnes valeurs (et les bons plats).
- On vérifie que la valeur répondue correspond à ce que l'on attends.

Copy

```
MenuDto newMenu = new MenuDto(  
    null,  
    "C'est l'anniversaire de Damien aujourd'hui!",  
    new HashSet<DishDto>(  
        Arrays.asList(  
            new DishDto(null, "Turkey"),  
            new DishDto(null, "Tiramisu")  
        )  
    )  
);  
  
Menu storedMenu = new Menu(  
    Long.valueOf(1),  
    "Christmas menu",  
    new HashSet<Dish>(  
        Arrays.asList(  
            new Dish(Long.valueOf(2), "Turkey", null),  
            new Dish(Long.valueOf(33), "Tiramisu", null)  
        )  
    )  
);
```

# Exercice: Écrire un test qui prouve que CreateMenuService réutilise les plats existants

## Plan de test

- Similaire au précédent, en revanche avant d'appeler `createMenu` on demande au `dishRepository` de répondre un `Dish` existant.
- On vérifie que l'instance de menu enregistrée référence bien le dish déjà créé

```
// Instancie un nouveau dish ayant pour identifiant 33.  
Dish existingDish = new Dish(Long.valueOf(33), "Tiramisu", null);  
  
// Configure le mock du dish repository pour retourner le dish existant  
// Quand il reçoit un appel à findByName avec la valeur ("tiramisu").  
when(dishRepository.findByName("Tiramisu")).thenReturn(existingDish);
```

Copy

# 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



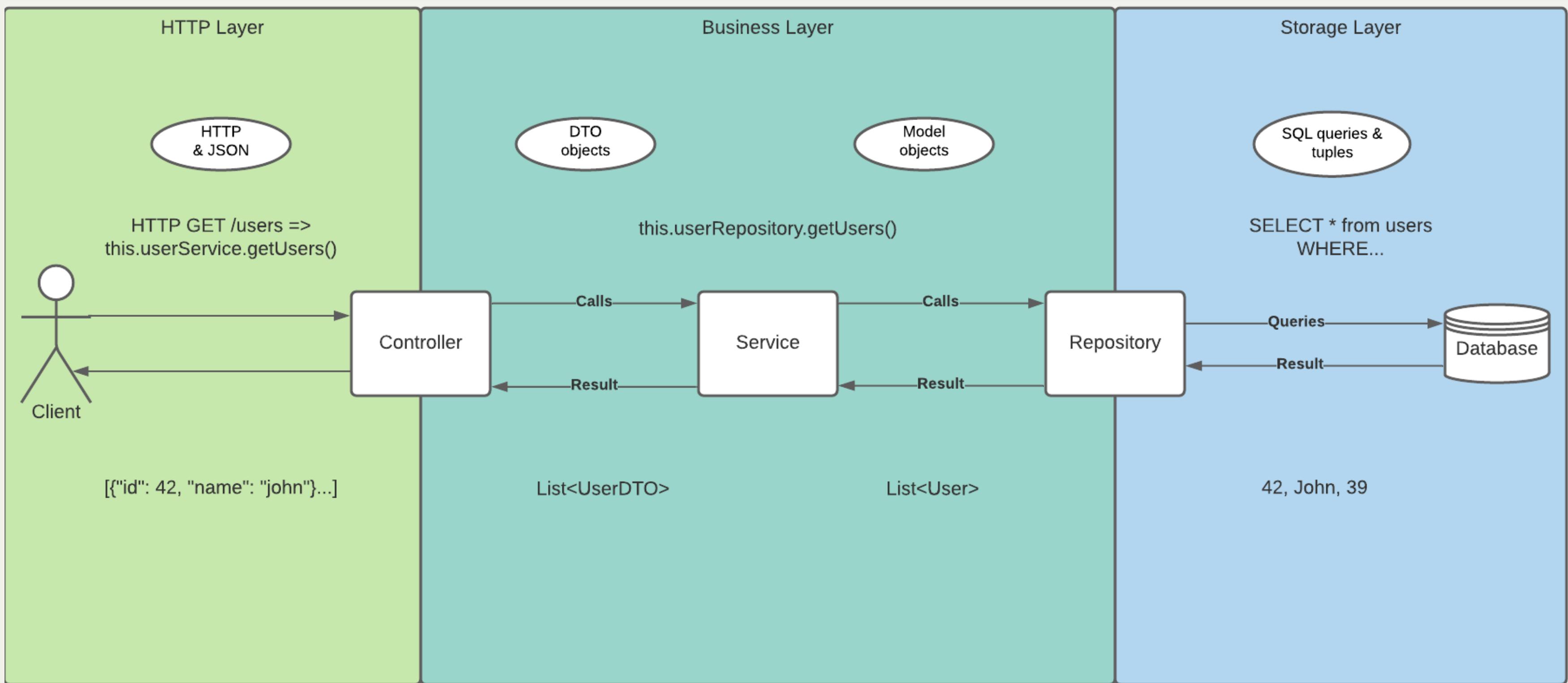


Tester des composants indépendamment ne prouve pas que le système fonctionne une fois intégré!

# 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 SUT (1/2)



# Définition du SUT (2/2)

Un test d'intégration doit à chaque test:

- 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.

# Configuration de Maven (1/2)

```
<plugins>
  <plugin>
    <!-- Configure le maven-surefire-plugin -->
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
      <!-- Désactive tous les tests pour l'exécution par défaut -->
      <skipTests>true</skipTests>
    </configuration>
    <executions>
      <execution>
        <!-- Crée une première exécution pour jouer les tests unitaires -->
        <id>unit</id>
        <phase>test</phase>
        <goals>
          <goal>test</goal>
        </goals>
        <configuration>
          <skipTests>${skipUnitTests}</skipTests>
          <!-- Inclue et execute les tests contenus dans les fichiers ayant le suffixe Tests.java-->
          <includes>
            <include>**/*Tests.java</include>
          </includes>
        </configuration>
      </execution>
      <execution>
        <!-- Crée une seconde exécution pour jouer les tests d'intégration -->
        <id>integration</id>
        <phase>integration-test</phase>
        <goals>
```

Copy

# Configuration de Maven (2/2)

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

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

# Mise en Place d'un Test d'Intégration

```
// 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 {

    @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

# Outillage fourni par SpringBoot

```
// Injecte automatiquement l'instance du menu repository
@Autowired
private MenuRepository menuRepository;

// Injecte automatiquement l'instance du TestRestTemplate
@Autowired
private TestRestTemplate template;

public void listExistingMenus() throws Exception {
    // Effectue une requête GET /menus
    ResponseEntity<MenuDto[]> response = this.template.getForEntity(getMenusURL().toString(), MenuDto[].class);

    //Parse le payload de la réponse sous forme d'array de MenuDto
    MenuDto[] gotMenus = response.getBody();
}
```

Copy

# Exercice: Implémentez le test d'intégration de GET /menus

- Provisionne la base de donnée avec des données fixes
- Effectue une requête HTTP sur GET /menus
- Parse la réponse sous forme de MenuDto
- Vérifie que le status de la réponse est 200.
- Compare la réponse à un résultat attendu de la même façon que dans le test unitaire.

# Exercice: Implémentez un test d'intégration pour DELETE /menus

- On crée un menu en base
- On fait un appel a DELETE /menus/{id}
- On vérifie que le menu n'existe plus en base
- On vérifie que les dishes du menu n'existent plus en base

# Checkpoint

On a vu :

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

# Encore plus d'intégration continue

C'est quand même le sujet du cours :)

# 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

# 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



4GIFs.com

# 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 à "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

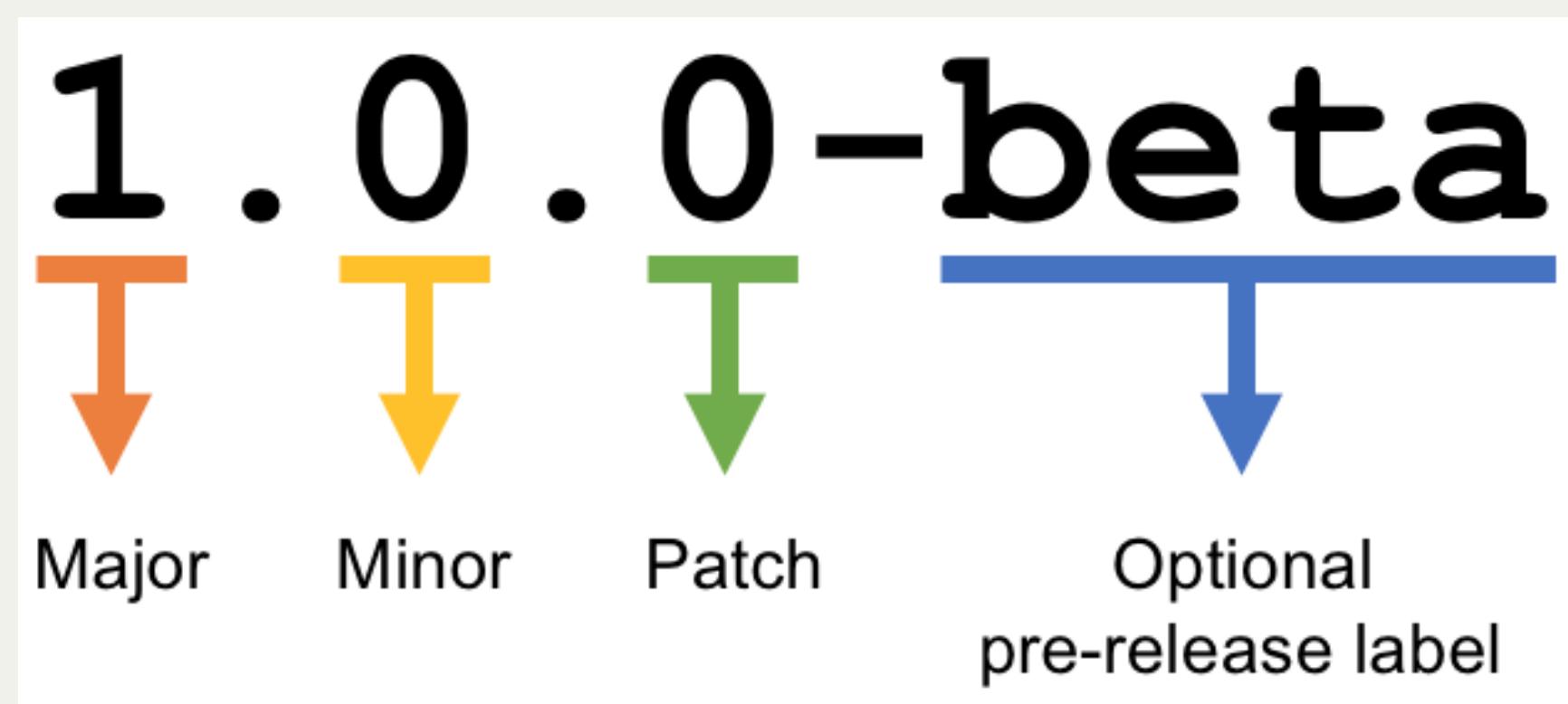
# Quoi versionner ?

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 v1.4.3 -m "Release version v1.4.3"
```

Copy

```
# Publier un tag sur le remote origin.  
git push origin v1.4.3
```

# "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!

# Anatomie d'une Release GitHub

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 a cette version (automatique)
  - Et éventuellement des fichiers de votre choix (des binaires compilés par exemple)

# Prérequis: Ordonnancement et exécution conditionnelle des jobs

- Un workflow exécute en parallèle, par défaut, tous les jobs qui le composent
- Il est possible de conditionner l'exécution d'un job à la terminaison d'un autre à l'aide du mot clé `needs` (documentation de `needs`)
  - Exécution séquentielle au lieu de parallèle
- 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 si build et test se sont terminés normalement.  
    needs: [build, test]  
    # 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 les actions fournies par GitHub:

- Action Download Artifact
- Create Release & Upload Release asset

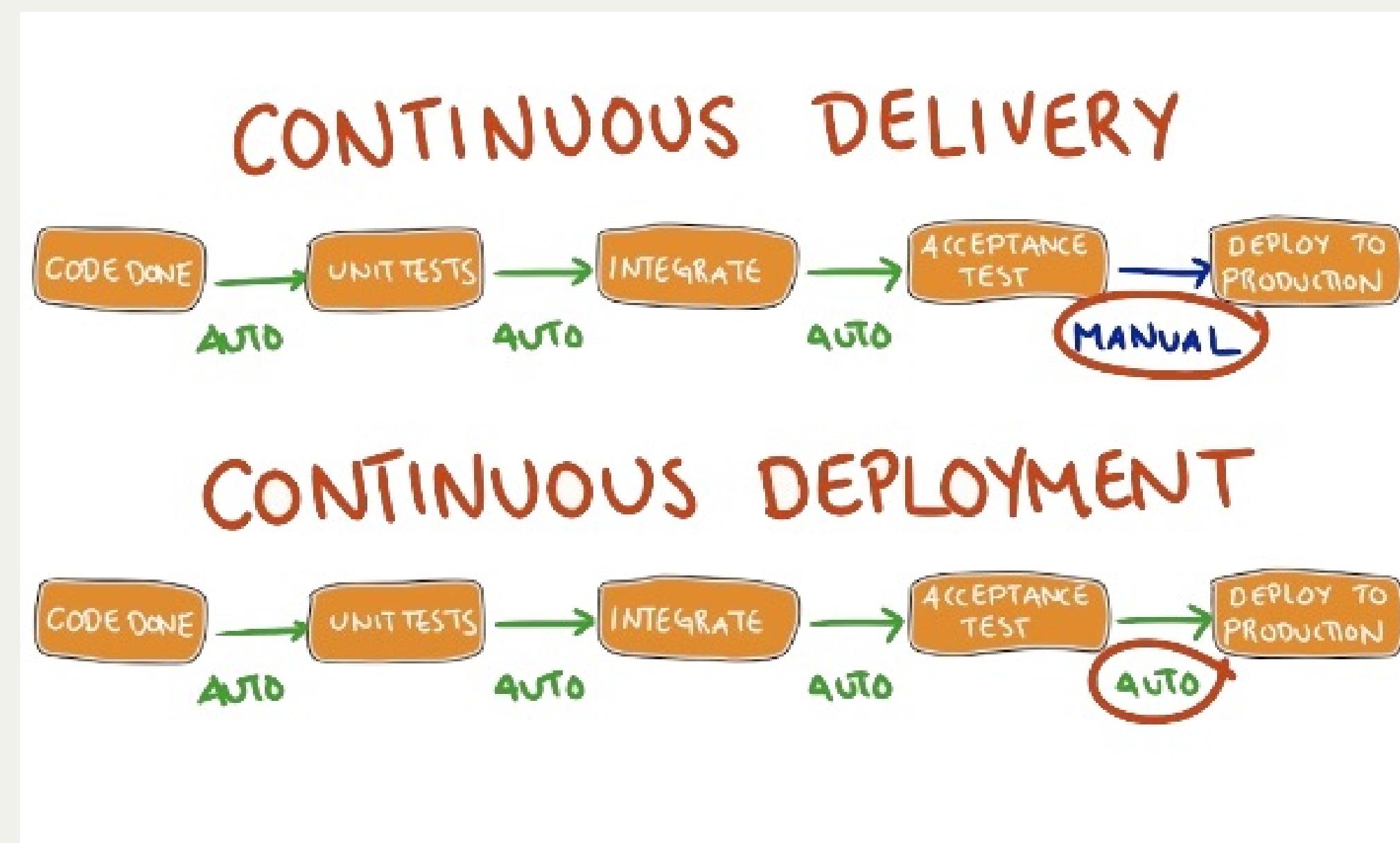
# Déploiement Continu

## 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 la ou 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 Heroku

- Dans le cadre de ce cours nous allons utiliser Heroku
- Plateforme d'hébergement automatisée
- Structurée autour de trois concepts principaux:
  - Un Dyno est une instance de votre serveur
  - Un Addon est un outil tierce dans votre application (un SGBD par exemple...)
  - Une Application est un ensemble de Dynos et d'Addons

# Exercice: Créez une application sur Heroku (1/2)

- Rendez vous sur Heroku et créez vous un compte
- Une fois sur la page Apps cliquez sur New app dans le menu en haut a gauche
- Sélectionnez un nom et une région (europe de préférence)
- Cliquez sur Create App

# Exercice: Créez une application sur Heroku (2/2)

- Rendez-vous sur l'onglet overview de votre app
  - Pour l'instance il ne s'y passe pas grand chose...
- Rendez vous maintenant dans l'écran "View Logs" en haut à droite
  - Permet de visualiser en temps réel les logs de vos Dyno en production
- Prenez un peu le temps de naviguer dans l'interface

# Déployer dans une App Heroku (1/2)

Heroku supporte plusieurs modes de déploiement

- Push du code source dans un remote heroku via git
- Connection directe avec GitHub
- Push d'une image dans une container registry
  - C'est cette méthode que nous allons utiliser (car la moins auto-magique)

# Construire une Image de Container pour Heroku

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

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

Copy

# Exercice: Créez et Lancez l'image Manuellement

- A la racine de votre dépôt menu-server créez un fichier Dockerfile
- Dans un terminal, lancez les commandes suivantes:

```
# Construit une image docker portant le tag `cicdlectures/menu-server:test`  
docker build -t cicdlectures/menu-server:test .  
  
# Lance un container basé sur l'image `cicdlectures/menu-server:test`  
docker run -ti --rm -e PORT=9090 -p 8080:9090 cicdlectures/menu-server:test  
  
# Vérifiez que vous pouvez faire des requêtes au menu-server....  
# Et Ctrl+C pour terminer l'exécution du container
```

Copy

# Déployer dans une App Heroku (2/2)

Les grandes étapes d'un déploiement dans heroku via une container registry

1. Le client construit une image de container de l'application et la pousse dans une registry d'images heroku
2. Le client indique à heroku de déployer cette nouvelle image dans un dyno (heroku container:release <nom du dyno>)

# Exercice: Déployez manuellement votre application à l'aide de heroku CLI

```
# Authentifie votre instance gitpod auprès de heroku
heroku login -i

# Authentifie le démon docker de votre instance auprès de la registry heroku
heroku container:login

# On repackage l'app
mvn package

# On construit l'image du container heroku et on la publie dans la registry heroku pour le dyno web
heroku container:push web --app <votre-app>

# On déploie la nouvelle image dans le dyno:web
heroku container:release web --app <votre-app>
```

Copy

Mais le faire manuellement c'est pas du CD. Il faut le faire faire à github actions

# Parler à Heroku depuis GitHub Actions

- Problème: Seul votre utilisateur peut contrôler votre application heroku (et heureusement!).  
Comment le CI peut-il se faire passer pour votre utilisateur Heroku?
- Solution: Il faut fournir au moteur de CI un token d'authentification (une chaîne de caractères) qui autorise le moteur de CI à prendre votre identité.
- ⚠ Ce token est une donnée sensible: si on vous la vole on peut se faire passer pour vous auprès d'Heroku
- Il faut donc le stocker en sécurité: sous forme de secret dans GitHub.

# Exercice: Récupérez votre Token Heroku et stockez le en tant que secret dans Github

- Rendez vous dans vos paramètres de comptes Heroku
- Dans la section API Key cliquez sur Reveal: cela affiche votre clé d'API
- Copiez cette valeur dans le presse papier
- Rendez vous maintenant dans la page de configuration de votre dépôt, section Secrets
- Cliquez sur New Repository Secret
- Entrez comme nom de secret HEROKU\_API\_KEY
- Collez la valeur (attention aux espaces!) et cliquez sur Add Secret

# Exercice: Mise en Place du Déploiement Continu dans 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-heroku` soit lancé
- Ce job effectue dans l'ordre:
  - Télécharge l'artefact de l'étape `build`
  - Checkout le depot (pour rapatrier le Dockerfile)
  - Effectue un déploiement sur heroku, en utilisant la CLI `heroku`

# Quelques astuces:

- La CLI heroku est déjà pré-installée dans votre environnement de CI
- On ne peut pas jouer `heroku login` car c'est une étape interactive.
  - En revanche la CLI utilise aussi la variable d'environnement `HEROKU_API_KEY` pour s'authentifier: il faut donc lui exposer le secret (doc!)
- Il ne reste plus qu'à jouer les commandes de déploiement que l'on a déjà vu.

# Pour aller plus loin...

- <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>

# Conclusion

# TODO

# Merci !

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

Slides: <https://cicd-lectures.github.io/slides/2021>



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