

# Introduction au CI/CD

ENSG - Décembre 2020

- Présentation disponible à l'adresse: <https://gifted-albattani-3bc8a1.netlify.app/>
- 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

- On a essayé de s'adapter à la situation et avons essayé de faire quelque chose d'interactif
- Il y aura donc une alternance de théorie et de pratique
- C'est la première fois qu'on le donne, il risque d'y avoir des soucis, be kind :-)
  - N'hésitez pas à ouvrir des PRs si vous en voyez **ici** (😉 wink wink)

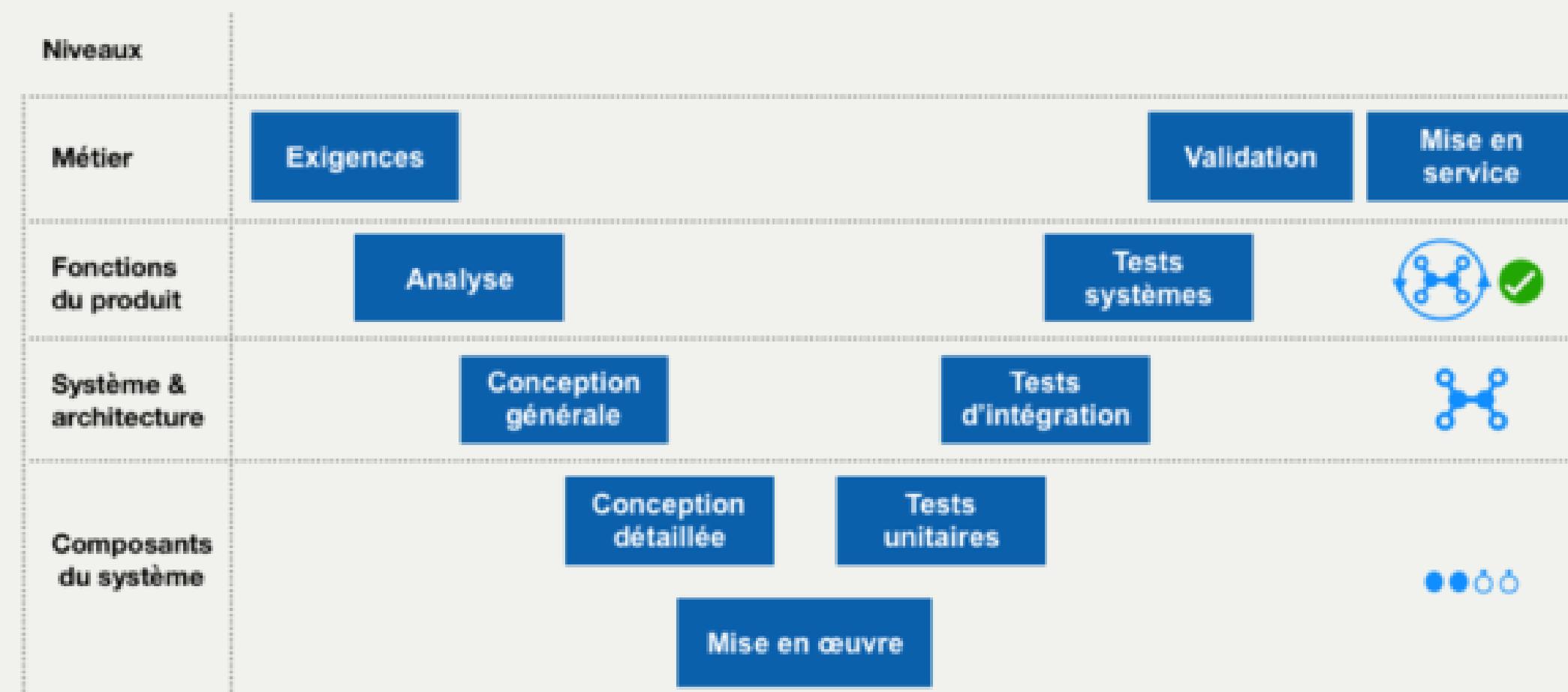
# Outils Necessaires

- Un navigateur récent (et décent)
- Un compte GitHub
- Un compte GitPod.io, notre environement de développement
- On va vous demander de travailler en binôme, commencez à réfléchir avec qui vous souhaitez travailler !

Ça veut dire quoi créer un logiciel ?

Savoir ce que l'on veut faire !

# Avant : le cycle en V



# What went wrong here?

- 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 fonctionel.
  - Réduire le périmètre fonctionel 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 aujourd'hui!

Disclaimer: les outils c'est une chose,  
l'important c'est le problème derrière!

# Changer du code

# Tracer le changement dans le code

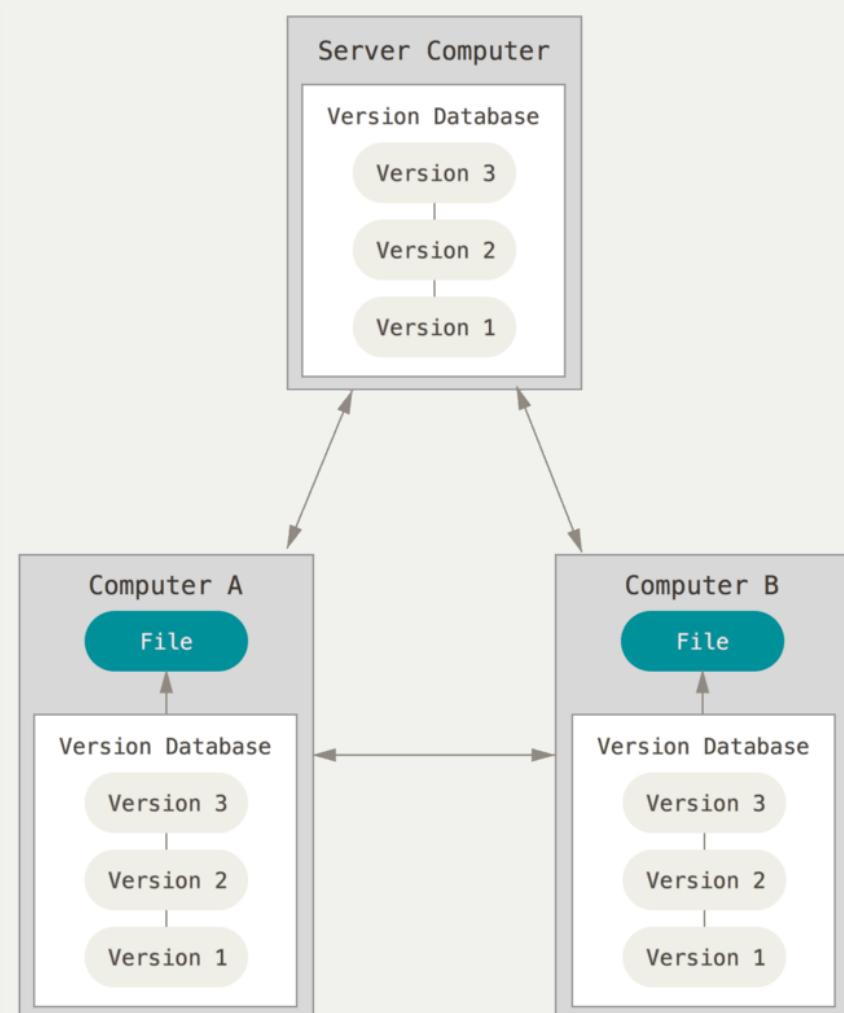
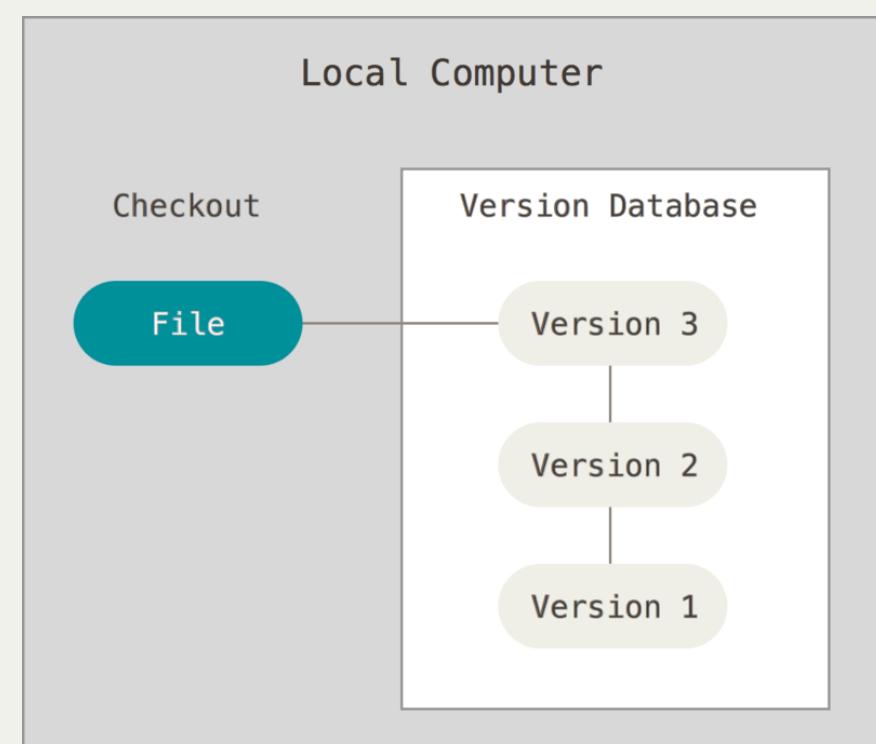
avec un **VCS** :  Version Control System

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

# Pourquoi un VCS ?

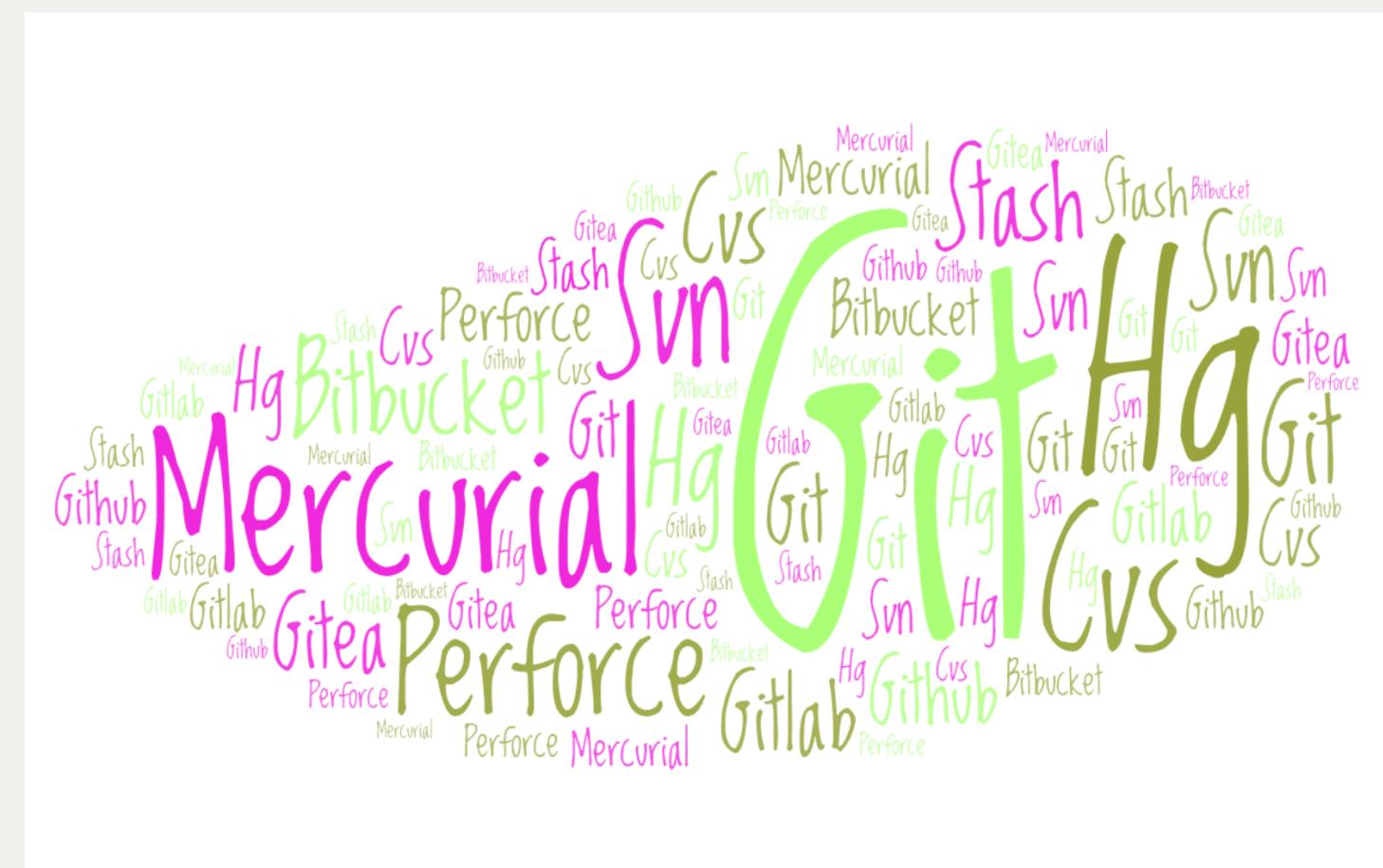
- Pour conserver une trace de **tous** les changements dans un historique
- Pour **collaborer** efficacement sur un même référentiel de code source

# Concepts des VCS



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

# Quel VCS utiliser ?



Nous allons utiliser **Git**

# Git

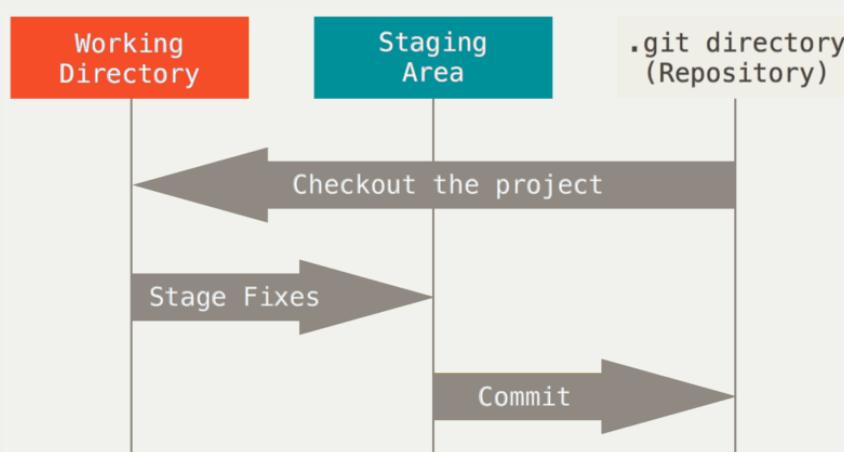
*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*

<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

- Accédez à l'environnement de travail
- Rendez vous dans le répertoire /workspace (cd /workspace)
- Créez un dossier vide nommé projet-vcs-1 puis positionnez-vous dans ce dossier

```
mkdir -p ./projet-vcs-1/ && cd ./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

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

**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

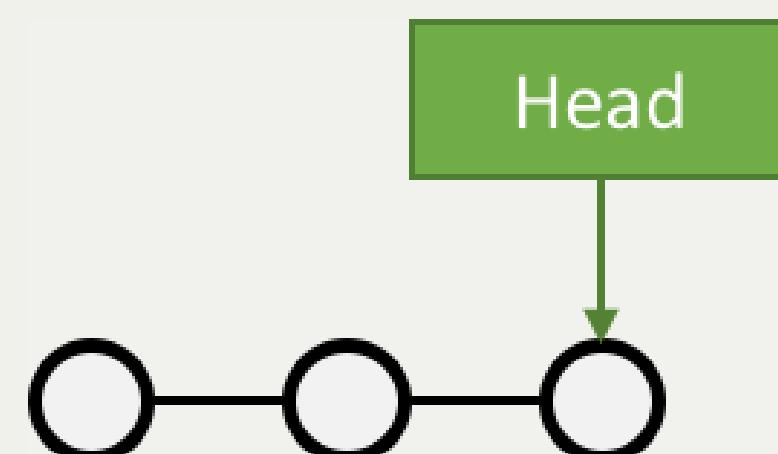
# Terminologie de Git - Commit

**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, e64ebe0, and its commit hash 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

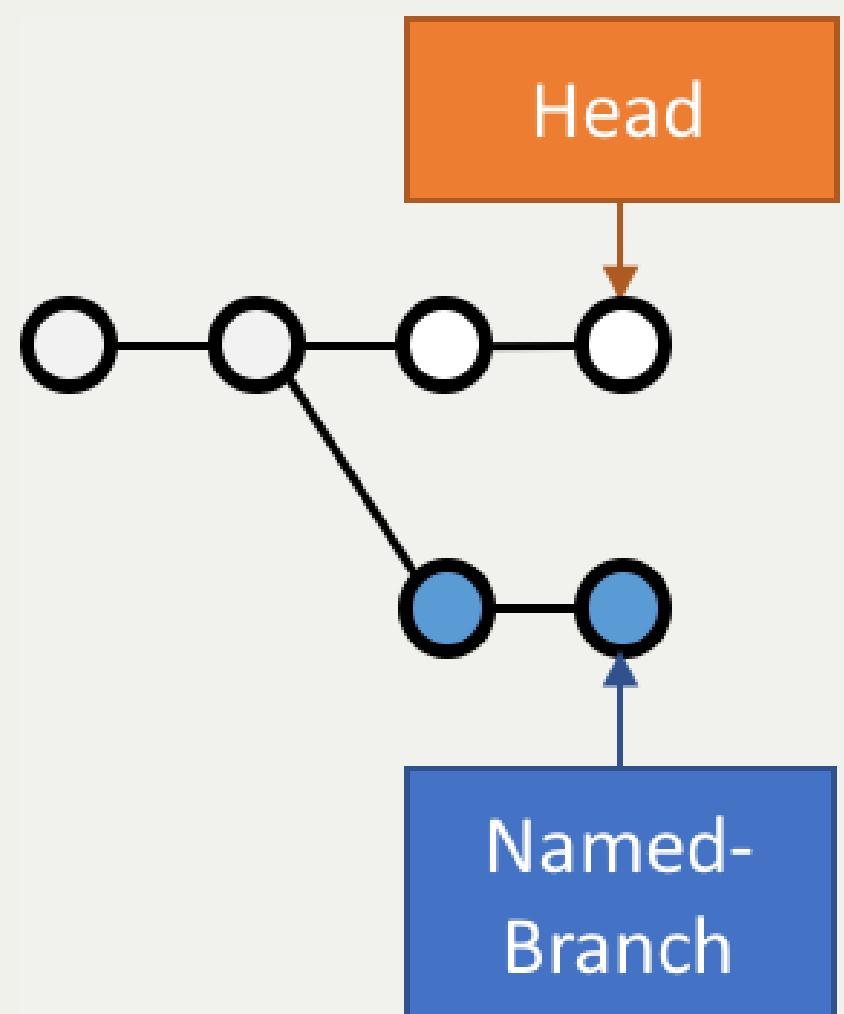
# Solution de l'exercice avec Git - 2

```
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`

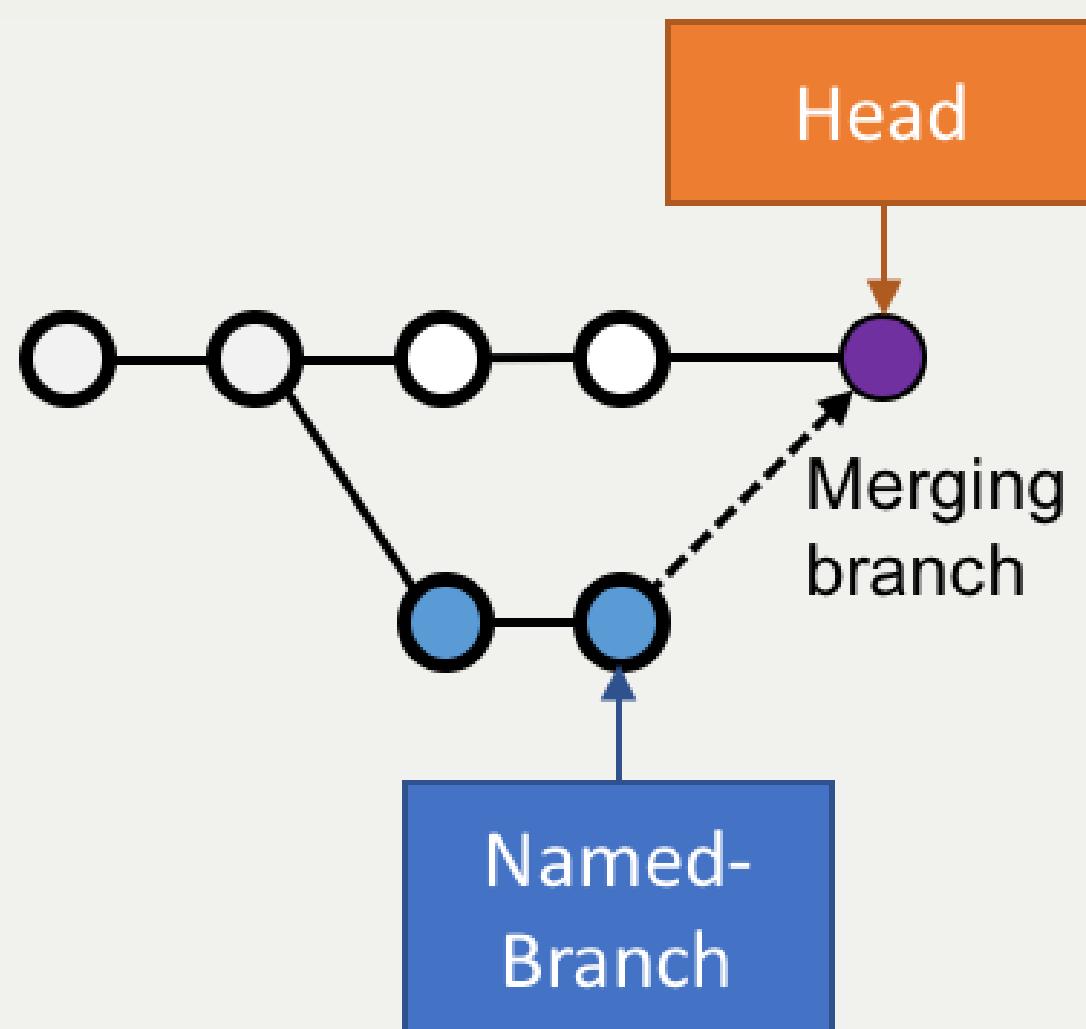
# Solution de l'exercice avec Git - 3

```
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`

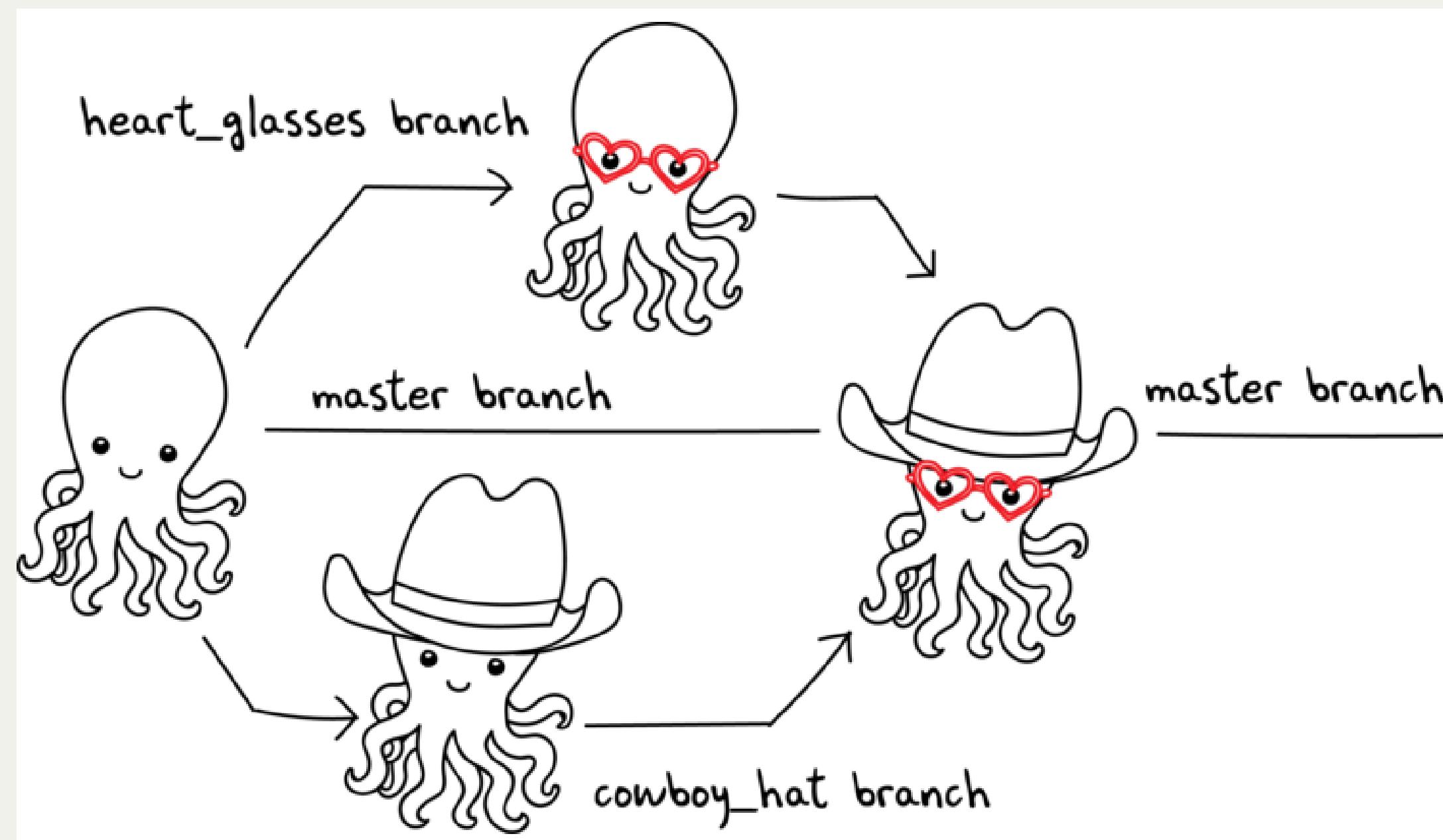
# Solution de l'exercice avec Git - 4

```
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/>

# Tests automatisés

Faire en sorte qu'on ne casse pas tout quand on change quelque chose...

# Qu'est ce qu'un test ?

- C'est du code qui vérifie qu'un système fait ce qu'il est supposé faire.
- Ecrire des tests est un acte préventif et non curatif.

# Pourquoi faire des tests ?

- Ça sert à prouver que le logiciel se comporte comme attendu à tout moment.
- Ça permet de détecter au plus tôt les problèmes.

# 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, différents types de tests

- Test unitaire
- Test d'intégration
- Test de bout en bout

# Test unitaire

- Test validant le bon comportement une unité de code (fonction, méthode...)
- 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

# Test Unitaire : Mise en place

- Accédez à l'environnement de travail
- Naviguez dans le projet /workspace/demoapp
- Passez sur la branche ut-exercise-1 (`git checkout ut-exercise-1`) et rafraîchissez l'explorateur de fichier
- Exécutez les tests unitaires, dans le terminal en bas, avec la commande `mvn test`
  - Spoiler : **✗**

# Test Unitaire : Exercice 1

Implementez la méthode greet de la classe GreeterService

- Si l'âge de l'utilisateur est inférieur à 10, alors retourner "Hi"
- Si l'âge de l'utilisateur est entre 10 et 20, alors retourner "Hey"
- Si l'âge de l'utilisateur est supérieur à 20, alors retourner "Hello"

# La classe GreeterService

```
// src/main/java/com/cicdlectures/demoapp/user/GreeterService.java

public class GreeterService {

    public String greet(User user) {
        // TODO
    }

}
```

Copy

# La classe User

```
// src/main/java/com/cicdlectures/demoapp/user/User.java

public class User {

    private int age;

    public User(String name, int age) {
        //...
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }
}
```

Copy

# La classe GreeterServiceTests

```
// src/test/java/com/cicdlectures/demoapp/user/GreeterServiceTests.java
```

```
class GreeterServiceTests {  
  
    private GreeterService subject;  
  
    @BeforeEach  
    public void init() {  
        this.subject = new GreeterService();  
    }  
  
    @Test  
    void greetsUserWithAgeBelow10WithHi() {  
        // ...  
    }  
}
```

Copy

# Un exemple de test

```
// src/test/java/com/cicdlectures/demoapp/user/GreeterServiceTests.java
```

Copy

```
@Test
@DisplayName("greets user with age below 10 with Hi")
void greetsUserWithAgeBelow10WithHi() {
    // Instancier un nouvel utilisateur.
    User user = new User("John", 5);

    // Appeler la methode a tester.
    String got = this.subject.greet(user);

    // Vefier le r  sultat.
    assertEquals("Hi", got);
}
```

# Implémentation du premier cas

```
// src/main/java/com/cicdlectures/demoapp/user/GreeterService.java
```

```
public String greet(User user) {
    if (user.getAge() < 10) {
        return "Hi";
    }
    //...
}
```

Copy

# A vous de jouer pour les deux autres cas :)

```
@Test
@DisplayName("greets user with age between 10 and 20 with Hey")
void greetsUserWithAgeBetween10And20WithHey() {
    fail("Not implemented");
}

@Test
@DisplayName("greets user above 20 with Hello")
void greetsUserWithAgeAbove20WithHello() {
    fail("Not implemented");
}
```

Copy

# Test Unitaire : Solution Exercice 1

```
git checkout ut-exercise-1-solution
```

Copy

# Test Unitaire: Exercice 2, mise en place

```
git checkout ut-exercise-2
```

Copy

# Test Unitaire : Exercice 2

Implementez la méthode `createUser` de la classe `UserService` et sa suite de tests.

- Si un utilisateur avec le même nom existe déjà dans la base de données, alors on ne fait rien.
- Sinon on enregistre ce nouvel utilisateur dans la base de données.

# Base de données ?

```
// src/main/java/com/cicdlectures/demoapp/user/UserRepository.java
```

```
public interface UserRepository {  
  
    // Enregistre l'utilisateur en base de donnée.  
    public void saveUser(User user);  
  
    // Retourne l'utilisateur en base qui porte le nom passé en paramètre.  
    // Retourne `null` si aucun utilisateur portant le nom existe.  
    public User findByName(String user);  
}
```

Copy

# La classe UserService

```
// src/main/java/com/cicdlectures/demoapp/user/UserService.java

public class UserService {

    private UserRepository repo;

    public UserService(UserRepository repo) {
        this.repo = repo;
    }

    public void createUser(User user) {
        // Regarde si un utilisateur avec ce nom existe en base.

        // Sauvegarde l'utilisateur si l'utilisateur n'existe pas.
    }
}
```

Copy

# Comment tester uniquement la classe UserService ?

- Le UserService à besoin d'un UserRepository pour fonctionner.
- Cependant :
  - On ne veut pas valider le comportement du UserRepository.
  - Pire, on ne veut pas se connecter à une base de donnée pendant un test unitaire.

# Remplacer le UserRepository (1/3)

Solution : On fournit une "fausse implémentation" au service.

```
// src/test/java/com/cicdlectures/demoapp/user/UserServiceTests.java
```

```
private UserRepository repository;  
  
private UserService subject;  
  
@BeforeEach  
public void init() {  
    this.repository = mock(UserRepository.class);  
    this.subject = new UserService(this.repository);  
}
```

Copy

# Remplacer le UserRepository (2/3)

que l'on pilote dans les tests!

```
@Test
public void createsUser() {
    // Quand le repository reçoit l'appel findByName avec la valeur "foo"
    // Alors il retourne null.
    when(repository.findByName("foo")).thenReturn(null);
}
```

Copy

# Remplacer le UserRepository (3/3)

et on valide les interactions avec cette instance!

```
@Test
public void createsUser() {
    User user = new User("foo", 10);
    // [...]
    // Verifie que l'instance de repository a reçu saveUser avec l'objet user.
    verify(this.repository).saveUser(user);
}
```

Copy

# Résumé

```
@Test
@DisplayName("creates an user")
public void createsUser() {
    User user = new User("foo", 10);
    when(repository.findByName("foo")).thenReturn(null);

    subject.createUser(user);

    verify(this.repository).saveUser(user);
}
```

Copy

# A vous de jouer pour l'autre cas :)

```
@Test
@DisplayName("does not create a user if it already exists")
public void doesNotcreateUserIfAlreadyExist() {
    fail("not implemented");
}
```

Copy

```
// Un peu d'aide :)

// Retourne l'utilisateur passé en paramètre.
when(repository.findByName("foo")).thenReturn(user);

// Vérifie que la méthode saveUser du repository n'est
// jamais appelé avec l'instance user.
verify(this.repository, never()).saveUser(user);
```

Copy

# Test Unitaire: Solution Exercice 2

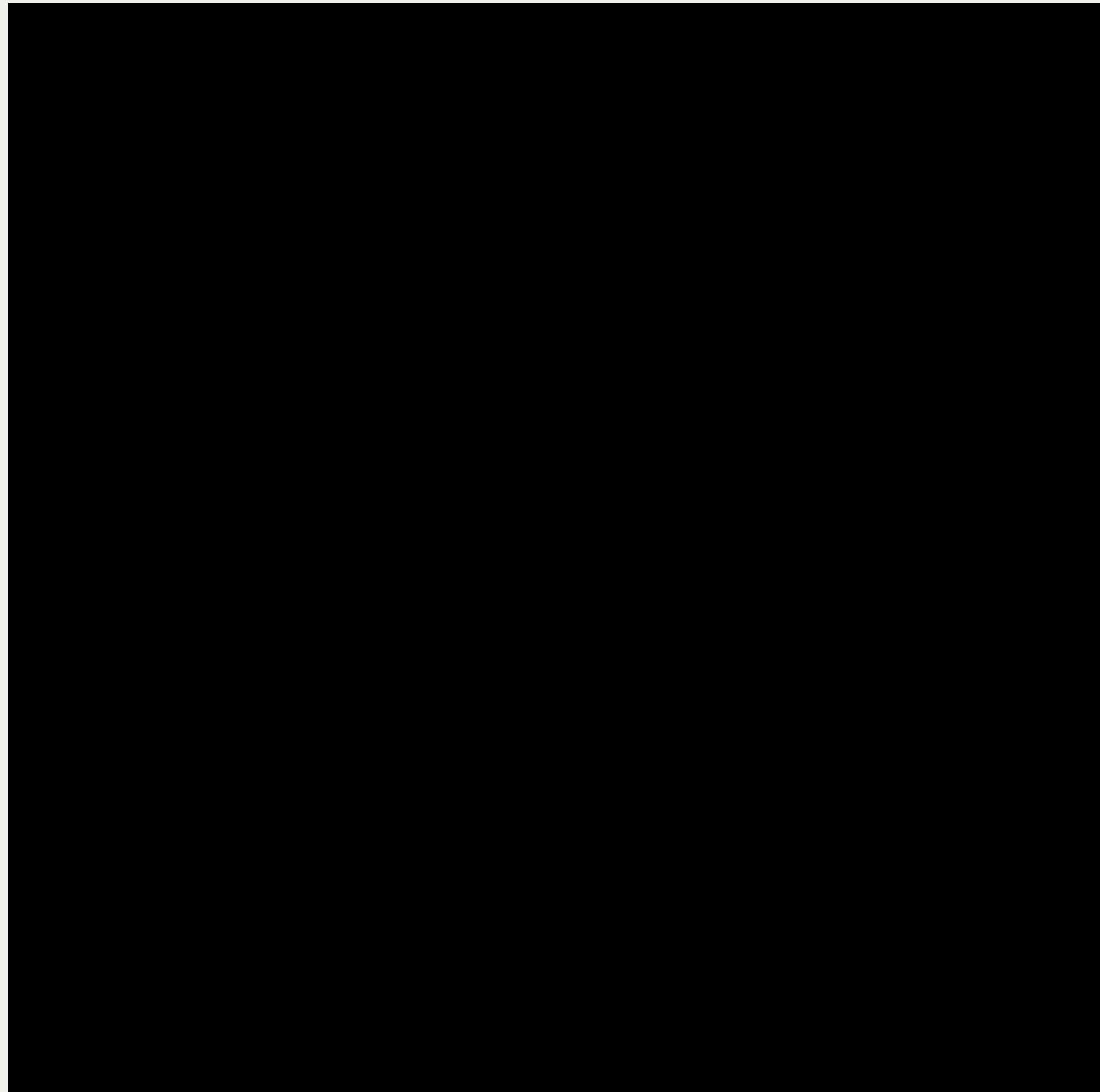
```
git checkout ut-exercise-2-solution
```

Copy

# 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

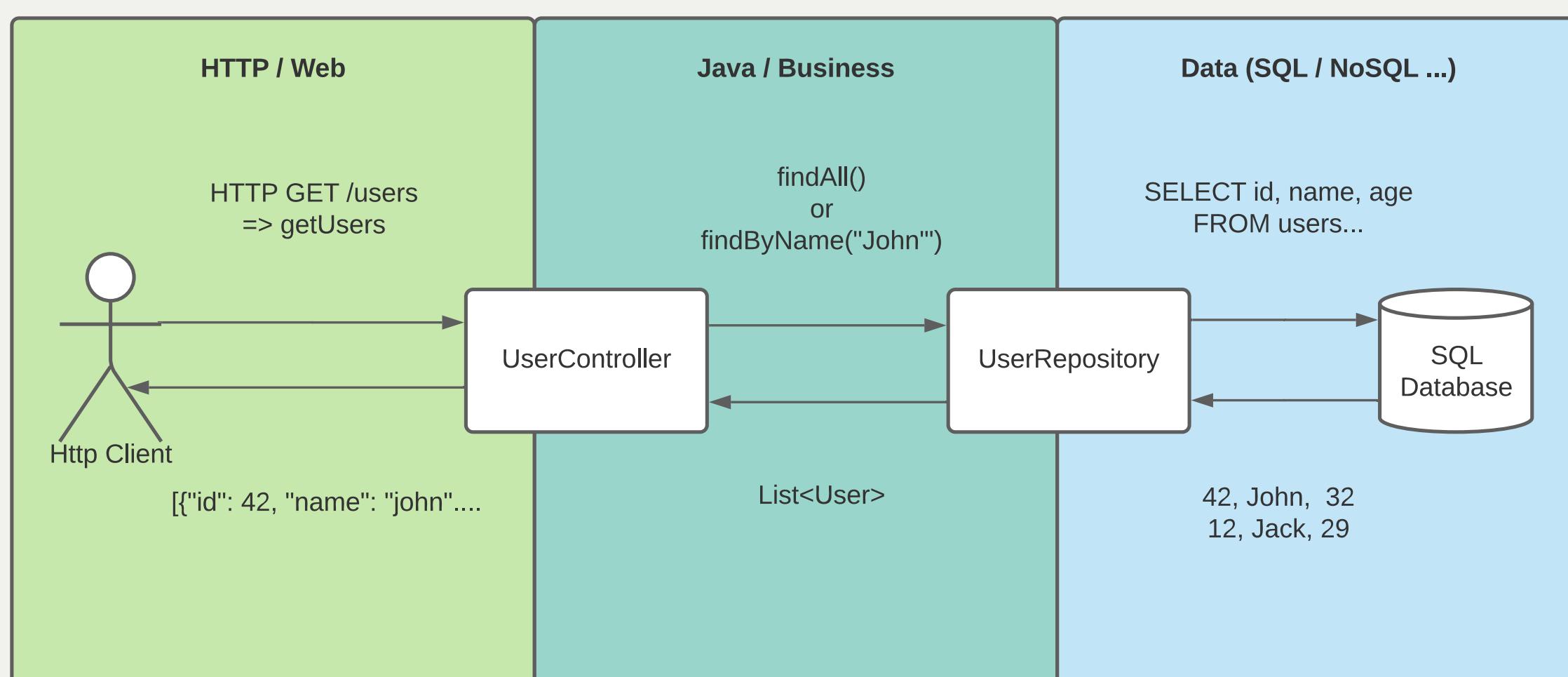




# Test d'Integration

- Test validant qu'un assemblage d'unités se comportent comme prévu.
- Par exemple :
  - Prouve que GET /users retourne la liste des utilisateurs en base

# Définition du Système à tester



(omission volontaire d'une couche de service a des fins de simplification)

# HTTP Client

Emet une requête HTTP et interprète la réponse.

Par exemple: curl, Firefox, Chrome, une autre app.

# UserController

Implémentation d'une requête HTTP par une methode java.

(en passant par un peu de magie spring)

- Parse les paramètres de la requête HTTP (headers, query parameters)
- Appelle la couche de données
- Réponds la donnée récupérée de la couche de données dans un format négocié.
  - HTML, JSON, XML ...

# UserRepository

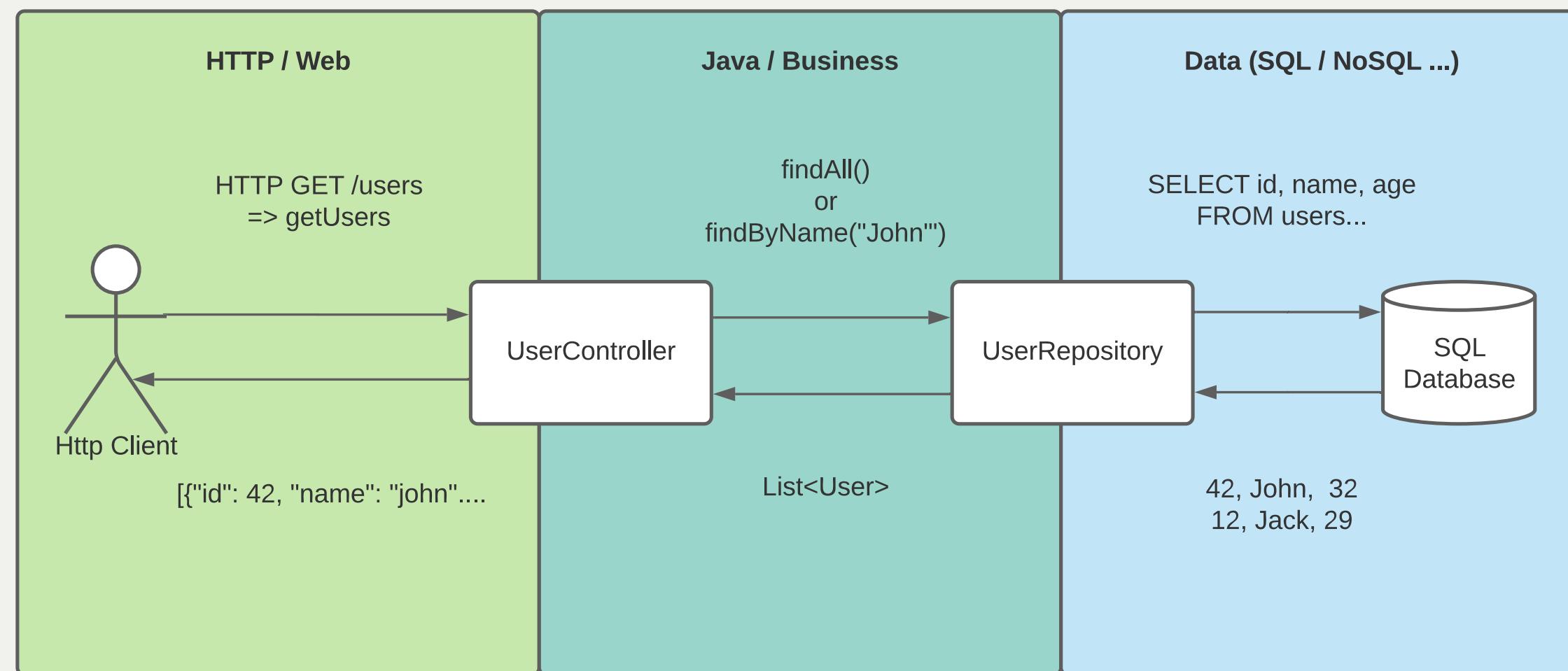
- Transforme un appel java en une requête à la base de données
  - SQL, PL/SQL, CQL, JSON (MongoDB, Elasticsearch...)
- Transforme la réponse de la base de donnée en objets java

# Base de données

- Reçoit des requêtes
- Réponds des données

Nous allons utiliser h2, une base de donnée SQL implémentée en java et s'exécutant en mémoire.

# En résumé



# De quel point de vue testons nous ?

Du point de vue du client HTTP.

# Exercice

Implémentez la méthode `getUsers` de la classe `UserController` et sa suite de tests pour qu'elle respecte le contrat suivant:

- Si le paramètre de requête "name" est vide alors on retourne tous les utilisateurs connus
- Si le paramètre de requête "name" non-vide alors on retourne la liste des utilisateurs ayant ce nom.

# Base de données 2, le retour !

```
// src/main/java/com/cicdlectures/demoapp/user/UserRepository.java
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {
    // CrudRepository fournit des méthodes de bases pour accéder à la donnée
    //
    // Par exemple:
    //
    // public Iterable<User> findAll();

    public List<User> findByNome(String nome);

}
```

Copy

# Interpréter et répondre à une requête HTTP

```
@RestController
public class UserController {

    @Autowired
    private UserRepository users;

    @GetMapping(path="/users", produces = "application/json")
    public Iterable<User> getUsers(@RequestParam(value = "name", defaultValue = "") String name) {
        // Appelle le user repository pour récupérer les données.
    }

}
```

Copy

# Notre client: le test!

```
// src/test/java/com/cicdlectures/demoapp/user/UserControllerIT.java

// Crée et initialise le serveur et le lance sur un port aléatoire.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class UserControllerTests {
    // [...]

    // Mets à jour l'attribut `url` avec le port du serveur (décidé aléatoirement).
    @BeforeEach
    public void setUp() throws Exception {
        this.url = new URL("http://localhost:" + port + "/users");
    }

    // Après chaque test, on vide la base de donnée.
    @AfterEach
    public void tearDown() throws Exception {
        this.userRepository.deleteAll();
    }
}
```

Copy

Copy

```
@Test
@DisplayName("lists all users")
public void testUsersList() throws Exception {
    // Définition du jeu de données.
    User[] wantUsers = {
        new User("John", 43),
        new User("Philip", 93),
        new User("Mitchell", 31)
    };

    // Enregistrement du jeu de données en base.
    for (User user : wantUsers) {
        this.userRepository.save(user);
    }

    // Appel HTTP GET /users sur l'URL du serveur lancé pour le test.
    ResponseEntity<User[]> response = this.template
        .getForEntity(url.toString(), User[].class);

    // Traitement et vérification des données de la réponse HTTP
}
```

# Implémentation du premier cas de test

```
public class UserController {  
  
    @GetMapping(path="/users", produces = "application/json")  
    public Iterable<User> getUsers(@RequestParam(value = "name", defaultValue = "") String name) {  
        return this.users.findAll();  
    }  
}
```

Copy

# A vous de jouer pour le second cas !

```
@Test
@DisplayName("filters users by name")
public void testUsersListFiltersByName() throws Exception {
    // Valide que lorsque le paramètre de requête name est non vide
    // Alors l'application répond les utilisateurs portant ce nom.
}
```

Copy

```
// Un peu d'aide :

// Vérifier qu'une chaîne de caractère est non vide
if (!name.isBlank()) {
    // ...
}

// Requête avec le paramètre de requête name à la valeur "Philip"
ResponseEntity<User[]> response = this.template
    .getForEntity(url.toString() + "?name=Philip", User[].class);
```

Copy

```
# Lancer les tests d'intégration
mvn verify
```

Copy

# Test d'Integration : Solution

```
git checkout it-exercise-1-solution
```

Copy

# Test d'Integration : Pro / Cons

- ✓ Relativement réalistes
- ✓ Potentiellement complexes
- ✓ Feedback "rapide"  $1s < t < 1m$
- ✗ Moins flexibles

# Cycle de vie technique

# Quel est le problème ?

On a du code (qu'on sait changer et tester).

- 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 qu'on utilise dans la "vraie vie"
- C'est versionné
- C'est *reproductible*

# Reproduire la fabrication

Comment fabriquer et tester de manière reproductible ?

- Linux / Windows / Mac
- Puissance disponible
- Habitudes différentes

⇒  Il faut des outils pour gérer le cycle de vie technique (Build → Test → etc.)

# Exemple avec Make

Fabriquons (joyeusement) des pages en HTML

# Exemple Make : Livrable

Définissons le livrable :

- Un dossier nommé dist...
- ...avec un fichier index.html dedans

```
$ ls ./dist/  
index.html
```

Copy

# Exemple Make : Code source

- On utilise le format **Asciidoc** pour écrire le **contenu**
  - Format de fichier .adoc
  - Exemple de syntaxe Asciidoc :

```
= Bonjour ENSG !  
  
An introduction to http://asciidoc.org[AsciiDoc].  
  
== First Section  
  
* item 1  
* item 2  
  
[source,bash]  
echo "Bonjour ENSG !"
```

Copy

# Exemple Make : Asciidoctor → HTML

⇒ C'est à vous (dans l'environnement GitPod)

- Créez un nouveau projet nommé exercice-makefile

```
mkdir -p /workspace/exercice-makefile && cd /workspace/exercice-makefile
```

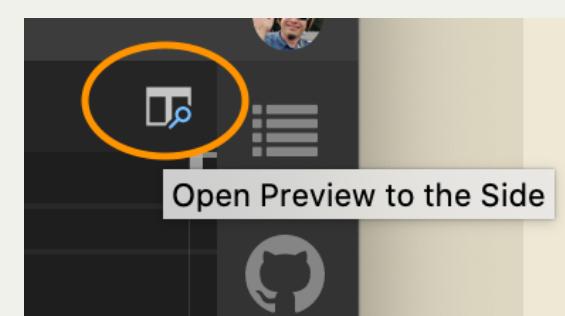
Copy

- Ajoutez un nouveau fichier main.adoc comme celui de la slide précédente
- Générez un fichier HTML avec la commande asciidoctor :

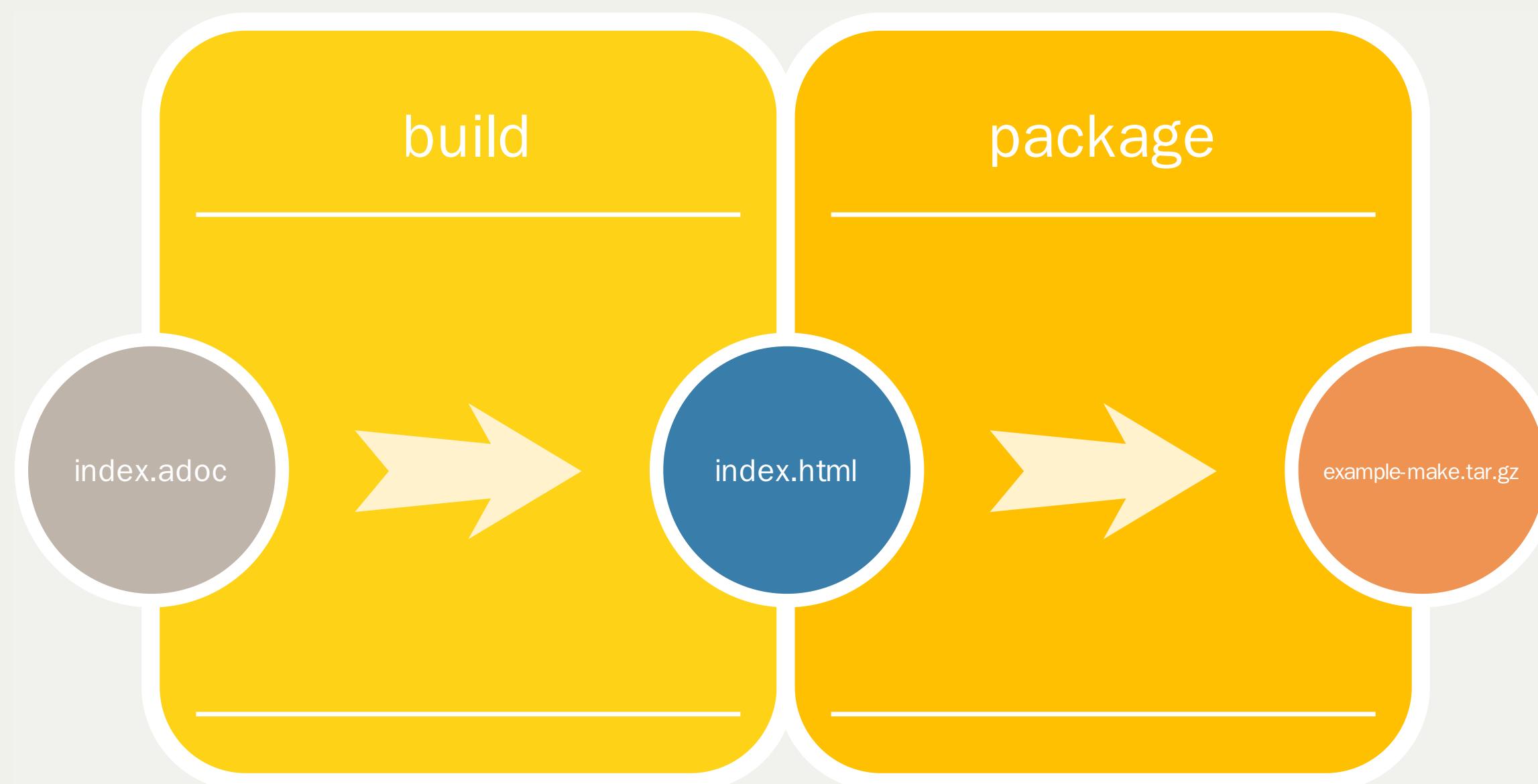
```
$ asciidoctor main.adoc
$ ls -ltr
# ...
main.html
# ...
```

Copy

- Affichez le fichier HTML en aperçu



# Exemple Make : Récapète



# Exemple Make : Introduction à Makefile

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

```
cible: dépendance
      commandes
```

Copy

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

```
make clean build
```

Copy

# Exemple de Makefile

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

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

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

Copy

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

Copy

# Exemple Make : Makefile - Build

- **But :** on veut générer le HTML en appelant la commande `make main.html`
- Toujours dans l'environnement GitPod, supprimez le fichier `main.html` si vous l'avez déjà
- Créez un fichier `Makefile` qui contient une cible `main.html` et qui appelle la commande `asciidoc`:

```
main.html:  
    asciidoc main.adoc
```

Copy

- Essayez avec la commande `make main.html`

# Exemple Make : Makefile Avancé

- Par défaut une cible/règle correspond à un fichier
  - Si le fichier existe, make ne ré-exécutera pas les commandes
- Pour désactiver ce comportement pour une cible donnée, ajoutez ladite cible comme dépendance à la cible spéciale .PHONY
  - On peut répéter .PHONY plusieurs fois
- Si vous appelez make sans argument, alors la cible par défaut sera la première cible définie

# Exercice Make : Makefile - Clean

- **But** : on veut supprimer les fichiers qu'on a généré lorsqu'on appelle la commande `make clean`
- Cette commande doit **toujours** s'exécuter, même si un fichier `clean` existe

⇒ C'est à vous !

# Solution : Makefile - Clean

```
main.html:  
    asciidoctor main.adoc  
  
.PHONY: clean  
clean:  
    rm -f ./main.html
```

Copy

```
touch clean # Créez un fichier "clean" bidon  
make main.html  
ls -l # 1 fichier "main.html"  
make clean  
ls -l # Aucun fichier "main.html"  
rm -f clean # Nettoyage
```

Copy

# Exercice Make : Makefile - Livrable

- **But :** on veut générer le livrable en appelant la commande `make dist`
- On utilisera les commandes `mkdir` et `cp` pour cette cible
- **⚠ Il y a une dépendance** sur la cible `main.html`
- **⚠ Il faudra sans doute adapter** `clean` et `.PHONY`

⇒ C'est à vous !

# Solution : Makefile - Livrable

```
main.html:  
    asciidoctor main.adoc
```

Copy

```
.PHONY: dist  
dist: main.html  
    mkdir -p ./dist  
    cp ./main.html ./dist/index.html  
  
.PHONY: clean  
clean:  
    rm -rf ./dist/ ./main.html
```

```
make clean  
ls -l # Aucun fichier "main.html" ni "dist/index.html"  
make dist  
ls -l # 2 fichiers "main.html" et "dist/index.html"
```

Copy

# Exercice Make : Makefile - All

- **But :** on veut que Make créé/re-créé le livrable **de zéro** par défaut lorsqu'on appelle make sans argument
- Cette cible doit s'appeler all et va appeler clean (au moins)

⇒ C'est à vous !

# Solution : Makefile - All

```
.PHONY: all
all: clean dist

main.html:
    asciidoctor main.adoc
```

Copy

```
.PHONY: dist
dist: main.html
    mkdir -p ./dist
    cp ./main.html ./dist/index.html

.PHONY: clean
clean:
    rm -rf ./dist/ ./main.html
```

Copy

```
make clean
ls -l # Aucun fichier "index.html" ou "example-make.tar.gz"
make
ls -l # 2 nouveaux fichiers "index.html" ou "example-make.tar.gz"
make clean
ls -l # Aucun fichier "index.html" ou "example-make.tar.gz"
```

# Exemple avec Maven

JAVA bien ?

# Exemple Maven : Livrable

- 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 Maven : Code source

- Code Source attendu dans `src//<language>/*`
  - Code Java de l'application dans `src/main/java/**`
  - Code (Java) des tests dans `src/test/java/**`

# Exemple Make : Java → JAR

- La commande `javac` permet de "compiler" du java en bytecode java:

```
$ javac ./src/main/java/com/cicdlectures/demoapp/Application.java --directory=./Application.class  
./src/main/java/com/cicdlectures/demoapp/Application.java:4: error: package org.springframework.boot.autoconfigure does not e
```

- Comment gérer les dépendances ?
- Comment assembler les `.class` pour faire un `.jar` ?
- Et les tests ?

⇒ C'est vite **complexe** alors qu'on veut écrire un programme

# Exemple Maven : Introduction à Maven

Say Hello to "Maven":

- Idée de Maven : **Cycle de vie** standardisé composé de "phases"
- Configuration Maven : `pom.xml` (Project Object Model)
- Ligne de commande `mvn` qui exécute les **phases**
  - Enchainement de **phases** séquentiellement

⚠ D'autres alternatives existent : Gradle, Bazel, etc.

# Exemple Maven : Commande mvn

Ligne de commande mvn :

- Lit le fichier `./pom.xml` pour comprendre le projet
- Exécute les phases ("étapes") passées en argument, ainsi que leurs dépendances :

```
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

- Le résultat est dans `./target`

# Exemple Maven : Les phases de Maven

- clean - Nettoie le contenu du dossier `./target`
- 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

# Exemple Maven : C'est à vous !

⇒ C'est à vous (dans l'environnement GitPod)

- Positionnez-vous dans le projet demoapp, sur la branche

```
cd /workspace/demoapp  
git fetch  
git checkout full-maven
```

Copy

- On va "nettoyer" ce qu'on a déjà produit:

```
mvn clean # Supprime le contenu du dossier `./target  
ls -l ./target # No such file or directory
```

Copy

- ⚠ La phase clean est spéciale et n'est jamais appellée par les autres phases

# Exemple Maven : Compiler

- But: Compilez le code source et visualisez le contenu du dossier target :

```
ls -l ./target # Avant  
mvn compile  
ls -l ./target # Après
```

Copy

# Exemple Maven : Tests Unitaires

- But: Compilez le code source et visualisez le contenu du dossier target :

```
ls -l ./target # Avant
mvn test
ls -l ./target # Après
ls -l ./target/surefire-reports
```

Copy

# Exemple Maven : Fabriquer le livrable

- But: Générez le livrable au format .jar :

```
ls -l ./target # Avant  
mvn package  
ls -l ./target # Après
```

Copy

# Exercice Maven : Tests d'intégration

- But: Exécuter les tests d'intégrations et afficher les rapport de tests
  - ⚠ Piège : regardez bien le contenu du dossier ./target

# Solution : Tests d'intégration

```
ls -l ./target # Avant  
mvn verify  
ls -l ./target/failsafe-reports
```

Copy

# Rendre son changement visible

Changer le code c'est bien. Mais à qui cela bénéficie-t-il ?

# Où votre changement est-il visible ?

- Un téléphone
- Les serveurs de votre client
- Vos propres serveurs
- Un microcontrôleur dans un satellite

# La Production

L'environnement où votre application est utilisée

# Différents environnement, différentes contraintes

Suite à un changement de code, il faut mettre à jour votre production.

- Est-ce qu'une mise à jour est facile ?
- Est-ce qu'une mise à jour va interrompre le service de production ?

⇒ Notion de risques liés au changement

# Gérer le risque

- Chaque changement comporte un **risque** : balance risque / bénéfice
- Diminuer le risque en validant un changement **avant** de l'exposer au reste du monde

# Valider un changement

- **Validation technique** : Est-ce que ça marche ? Est-ce que c'est assez rapide ?
- **Validation fonctionnelle** : Est-ce que ce qui a été réalisé correspond aux attentes?

# Comment aller jusqu'à la production ?

# Procédure jusqu'à la production

C'est facile ! Soyez Attentifs !

- Tests Unitaires : make unit-test
- Tests d'intégration : make integration-test
- Merger la branche : git merge
- Pousser les changements : git push
- Générer les artefacts : make jar
- Déployer les artefacts : scp ./jar-prod.jar  
monutilisateur@mamachinedeprod.com:/app/rbinks/jar.jar
- Redémarrer mon serveur : ssh monutilisateur@mamachinedeprod.com -C  
"systemctl restart binks"

# Qu'est-ce qui peut mal se passer ?

- Oublier / Inverser une étape
- Les tests n'ont pas été lancés depuis 3 semaines...
  - ...ils sont pétés, et on n'a pas le temps de fixer...
- Et si...
  - pas les droits d'accès ?
  - "cel•le•ui-ki-fait-ça-d'habitude" est malade ?
  - on a 10 ou 100 serveurs au lieu de 1 \o/ ?

# Une seule solution : l'automatisation !

*Si ça fait mal, il faut le faire souvent !*

- Rendre systématique le maximum d'opérations
- Automatiser les tâches redondantes

# Comment automatiser ?

Say "Hello" to CI

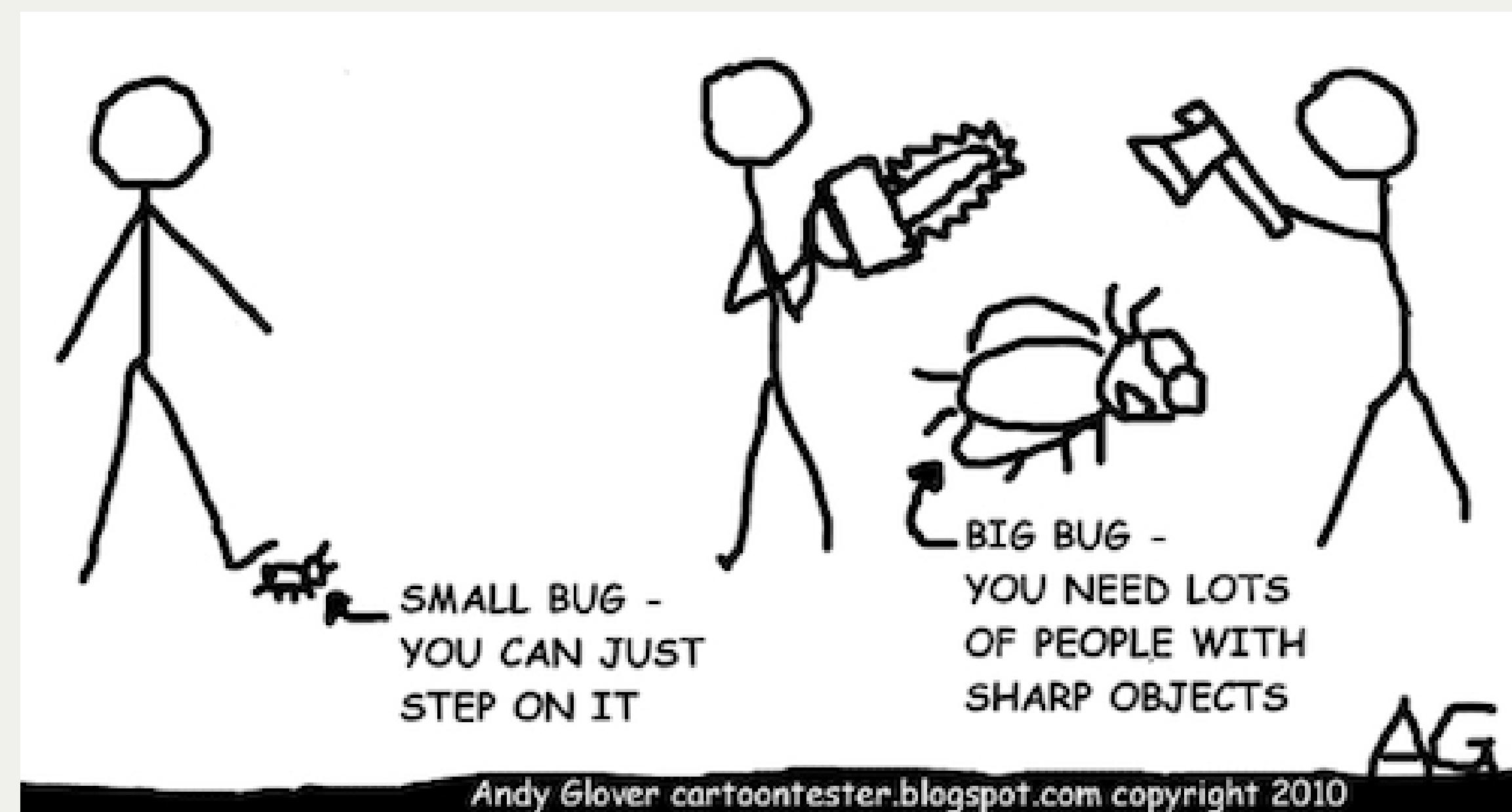
# Intégration Continue (CI)

*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



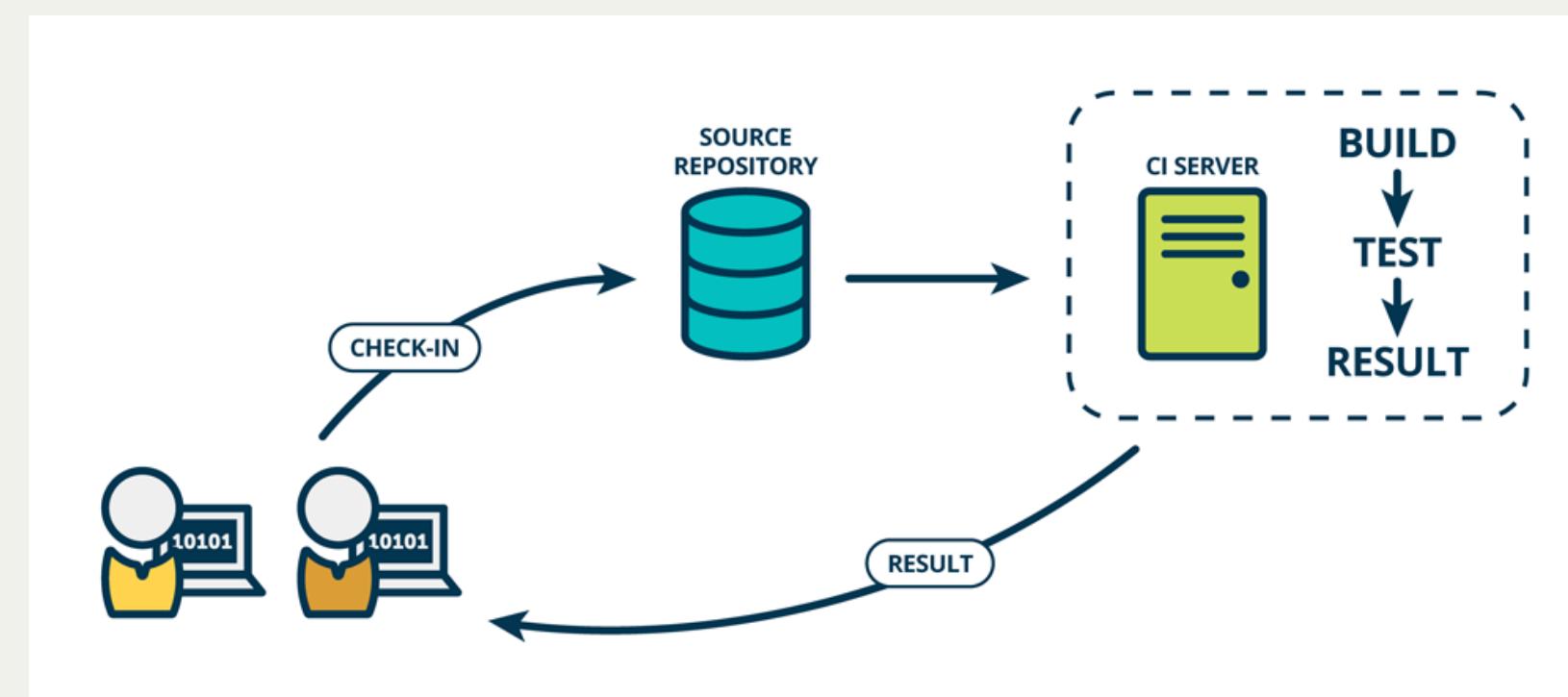
Source : <http://cartoontester.blogspot.be/2010/01/big-bugs.html>

# 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 développeu•se•r ajoute du code dans git : un évènement est transmis au "CI"
- Le CI compile et teste le code
- On ferme la boucle : Le résultat est rendu au développeu•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

# "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

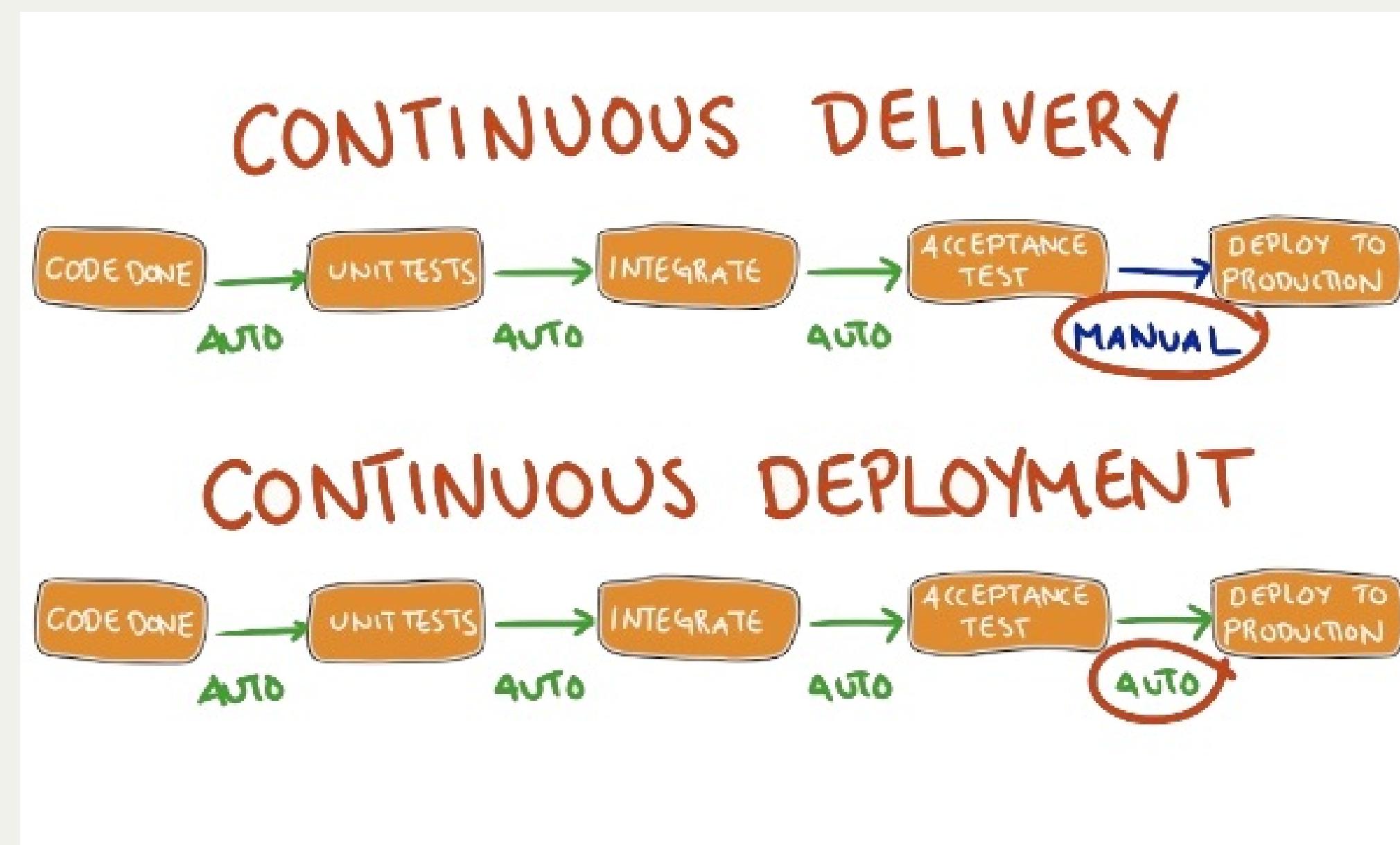
# Déploiement Continu

## Continuous Deployment

# Qu'est ce que le Déploiement Continu ?

- Version "avancée" de la livraison continue:
  - Chaque changement **est** déployé en production, de manière **automatique**
- Question importante: En avez-vous besoin ?
  - Avez-vous les mêmes besoin que Amazon Google ou Netflix ?

# Continuous Delivery versus Deployment



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

# Pour aller plus loin...

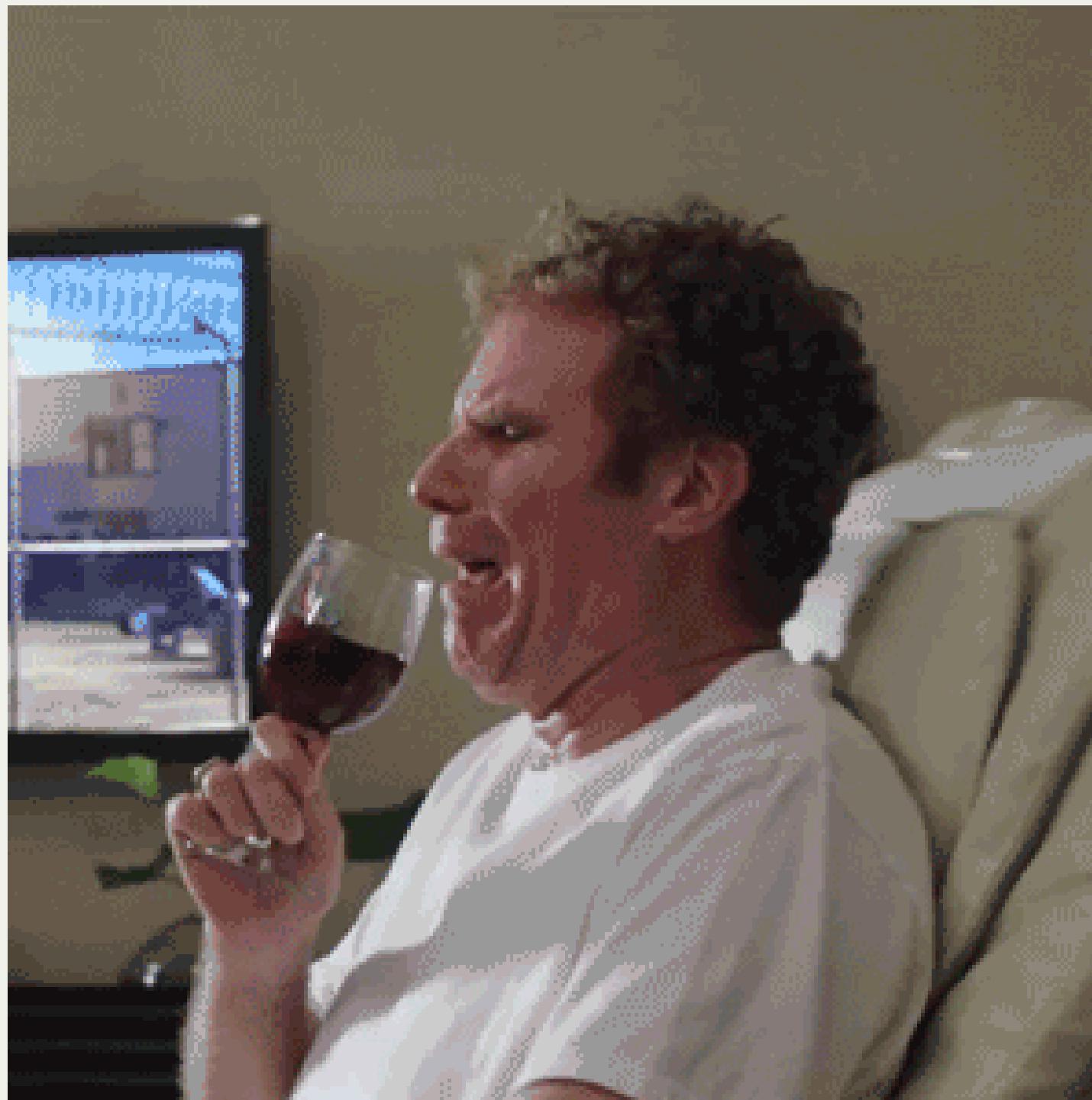
- <http://martinfowler.com/articles/continuousIntegration.html>
- <http://martinfowler.com/bliki/ContinuousDelivery.html>
- <https://jaxenter.com/implementing-continuous-delivery-117916.html>
- <https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/>
-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

# GitHub : Aller plus loin avec git

# 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 échapez votre tasse de thé / café sur votre ordinateur ?
  - Une météorite tombe sur votre bureau et fracasse votre ordinateur ?



Testé, pas approuvé.

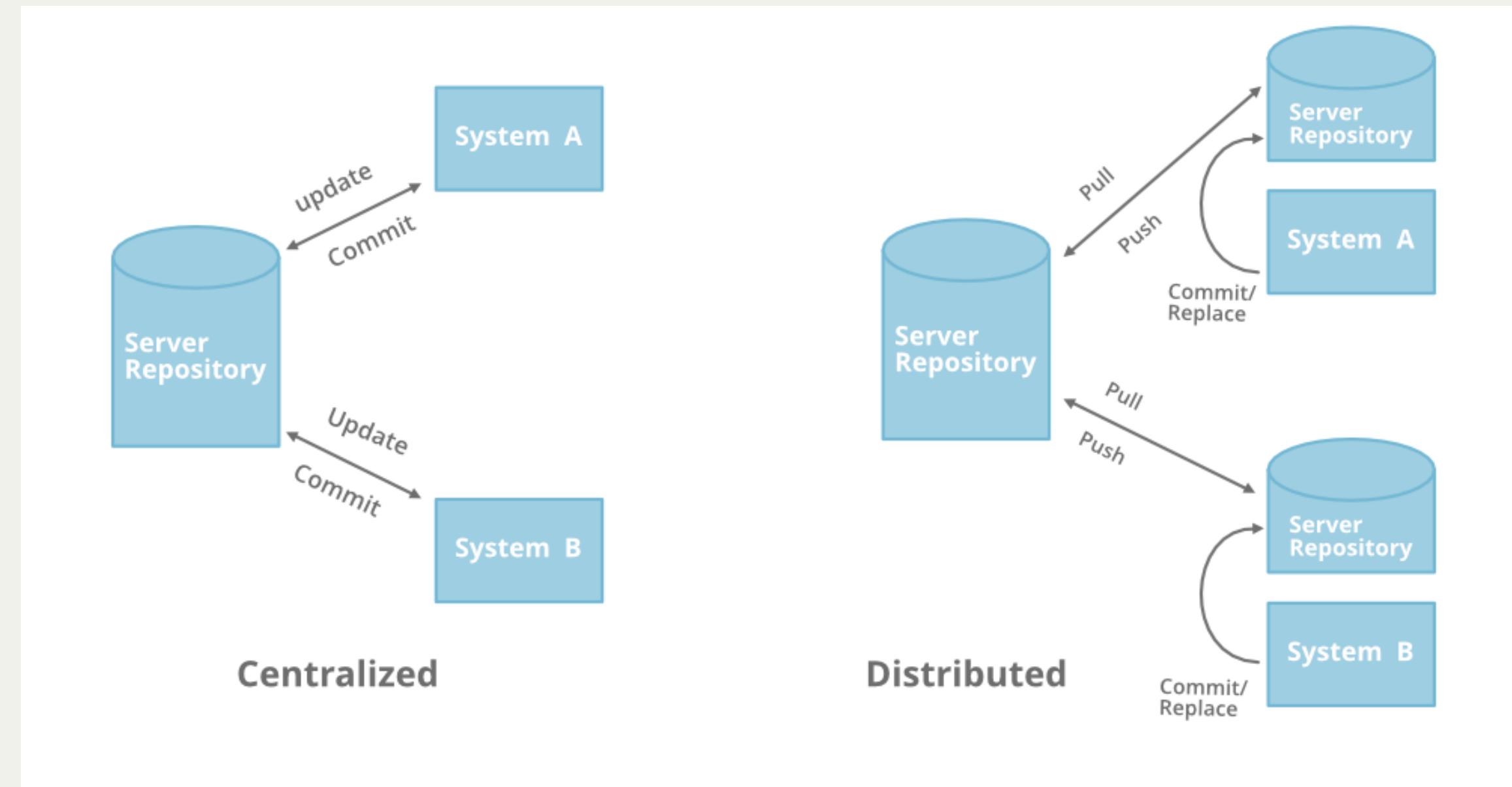
# Comment éviter ça ?

- Répliquer votre dépôt sur une autre machine !
- Git, comme tout autre CVS, peut gérer ce type de problème

# Centralisé vs Decentralisé

Git n'est pas un CVS, mais un DCSV

- Chaque utilisateur maintient une version du dépôt *local* qu'il peut changer à souhait
- L'opération de propager une version sur un dépôt **distant** est décorrelée du commit
- Un dépôt *local* peut avoir plusieurs dépôts **distant**.



Source Geek for Geeks

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

# Mise en place de l'exercice (1/2)

- Rendez vous sur la page des droits GitPod
  - Cochez dans la colonne github "write public repos" et "update workflows" puis validez
- Rendez vous sur Github
  - Créez un nouveau dépôt distant en cliquant sur "New" en haut à gauche
  - Une fois créé, mémorisez l'URL ([https://github.com/...](https://github.com/)) de votre dépôt :-)

# Mise en place de l'exercice (2/2)

Accédez à l'environnement de travail, puis depuis le terminal jouez les commandes suivantes:

```
cd /workspace/  
mkdir -p <Nom de votre dépôt>  
cd ./<Nom de votre dépôt>/  
# Initialize un nouveau dépôt git dans le répertoire  
git init  
# Crée un premier commit vide dans votre dépôt  
git commit --allow-empty -m "Initial commit"  
# Renomme la branche courante "master" en "main"  
git branch -m main
```

Copy

# Consulter l'historique de commits

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

```
# Ici il n'y en a qu'un seul!
```

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.
- C'est une 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

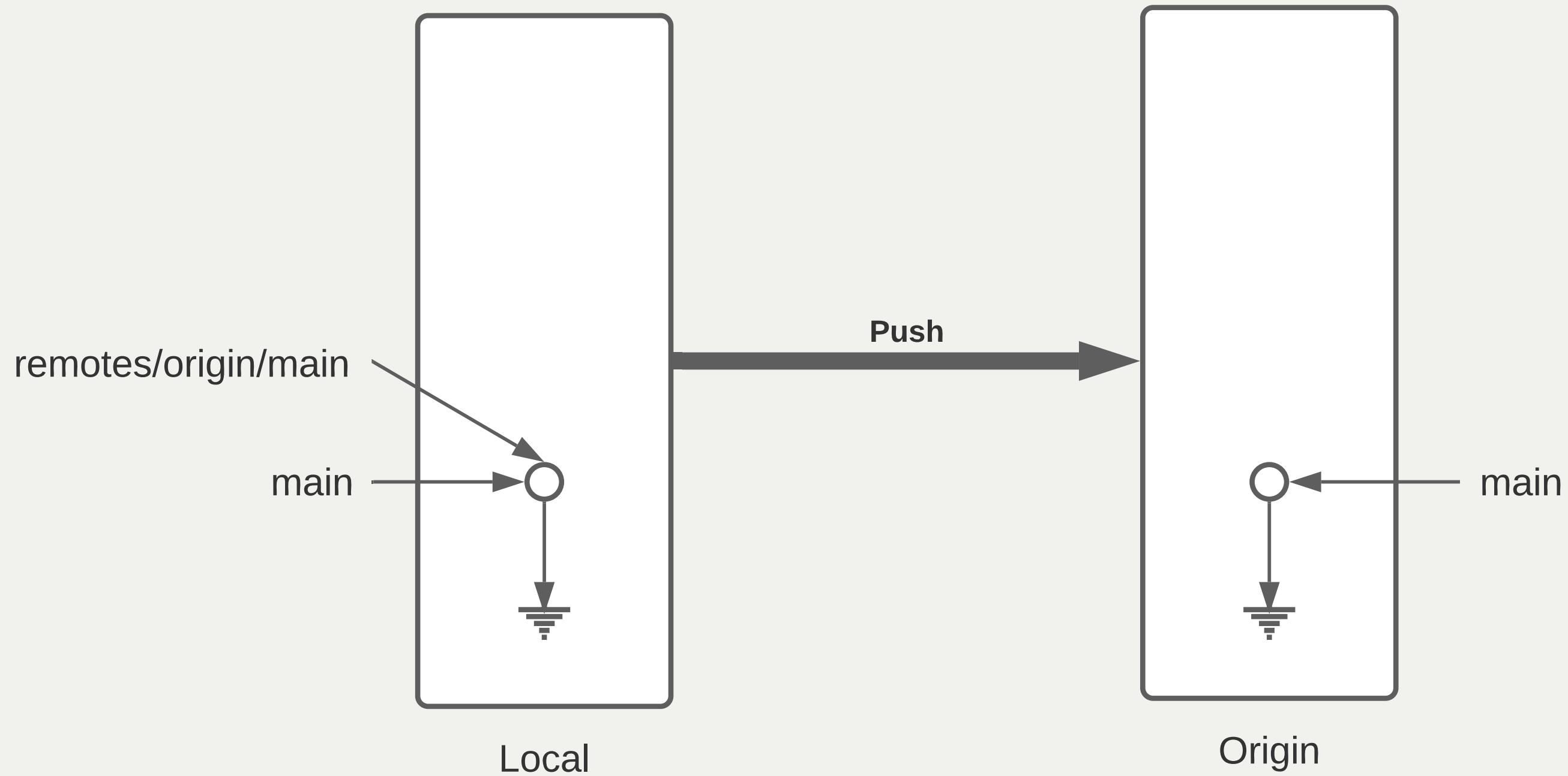
# Publier une branche dans sur dépôt distant

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 ?

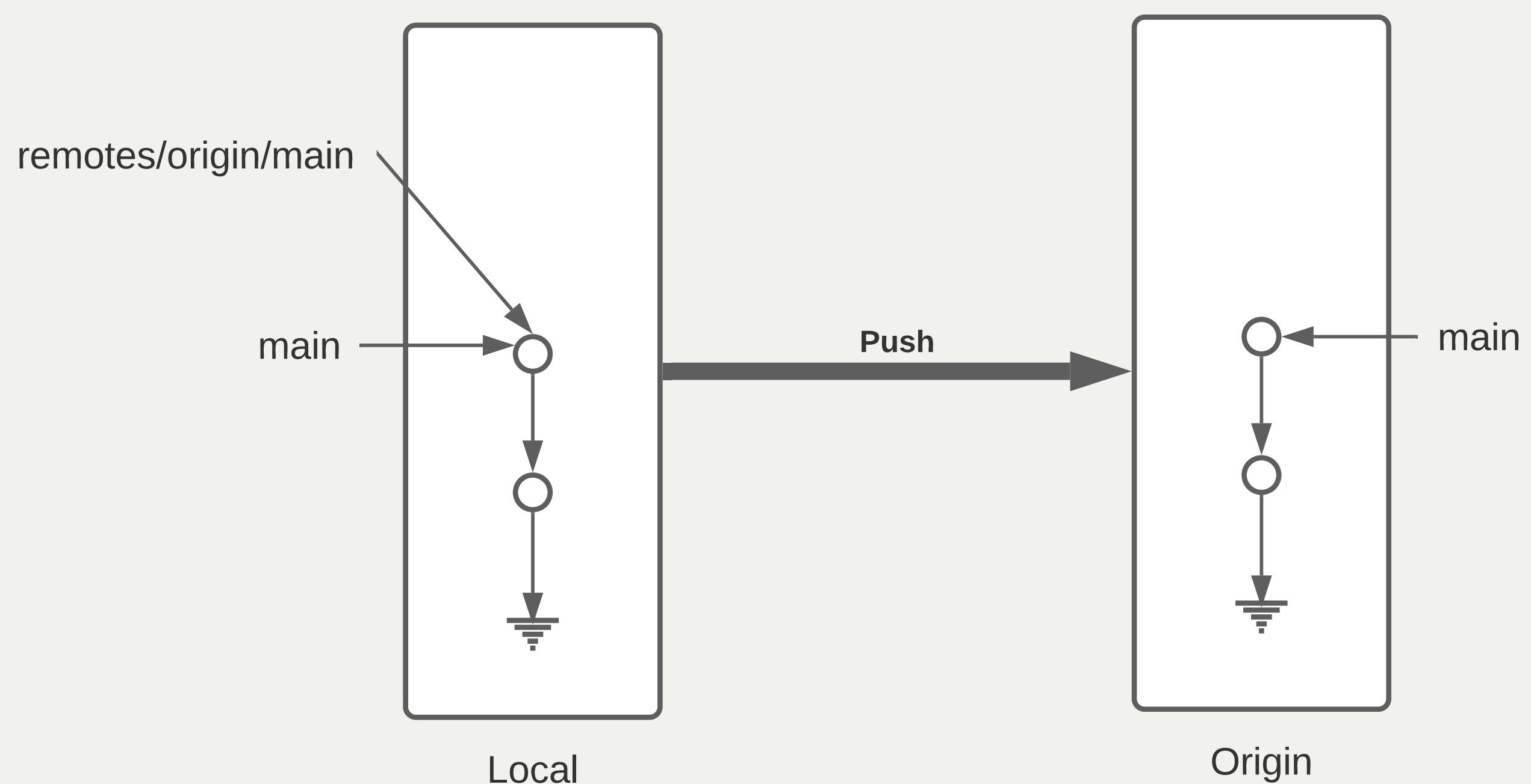


- git a envoyé la branche main sur le remote origin
- ... qui à accepté le changement et mis à jour sa propre branche main.
- git a créé localement une branche distante origin/main qui suis l'état de main sur le remote.

# 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 maintenue par git  
C'est une image du dernier état connu de la branche sur le remote.

Pour la mettre à jour depuis le remote il faut utiliser :

```
git fetch <nom_du_remote>
```

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

# Notez que est listé remotes/origin/main

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

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

Copy

# Créez un commit depuis GitHub directement

- Rendez vous sur la page de votre dépôt
- Cliquez sur "Add a README"
- Rajoutez du contenu a votre README
- Dans la section "Commit a new file"
  - 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

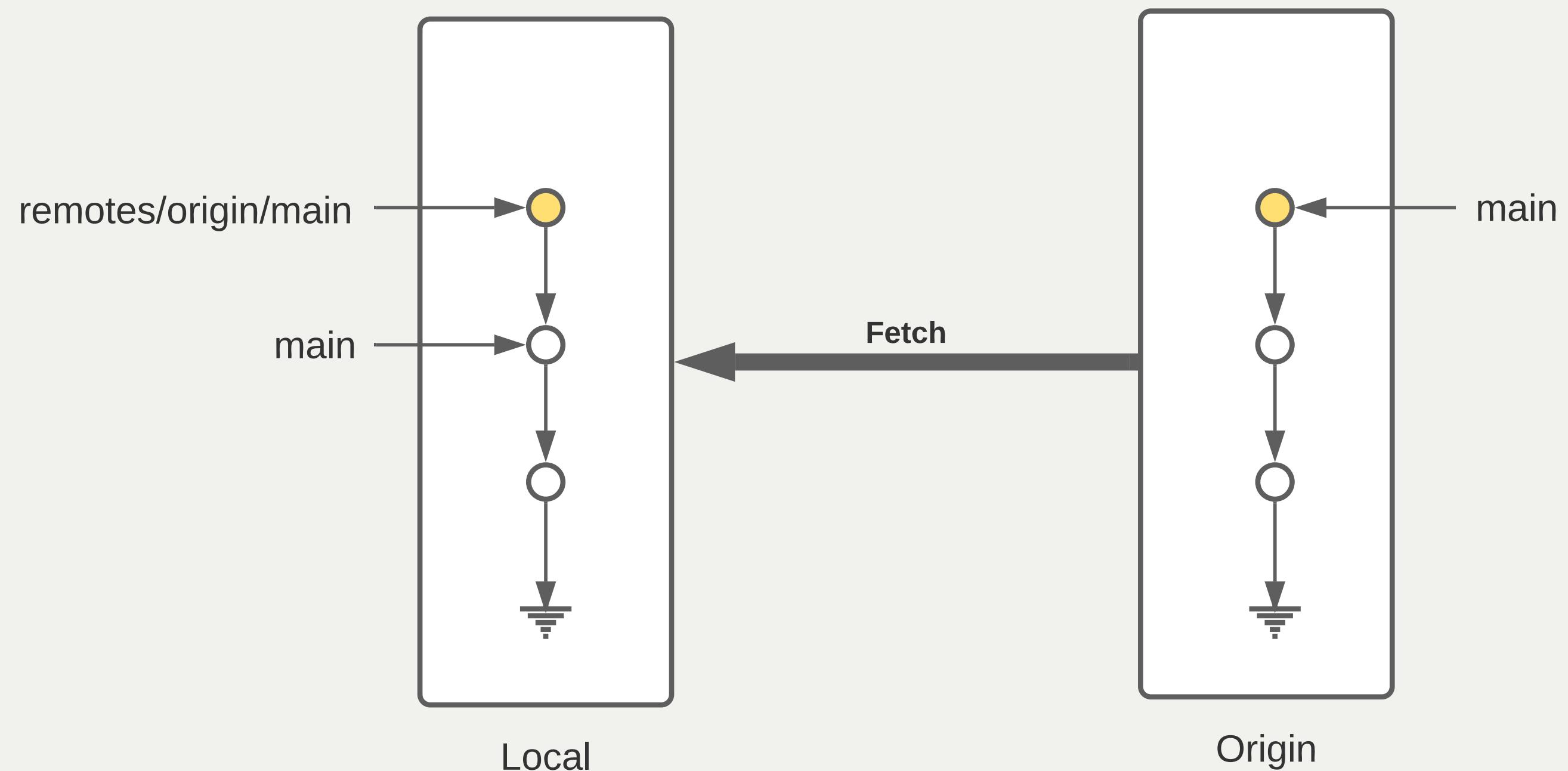
# La branche distante main a avancé sur le remote origin
# => La branche remotes/origin/main est donc mise à jour

# Listez les fichiers présents dans le dépôt
ls

# Mystère, le fichier README n'est pas là ?
# Listez l'historique de commit
git log

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

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
```

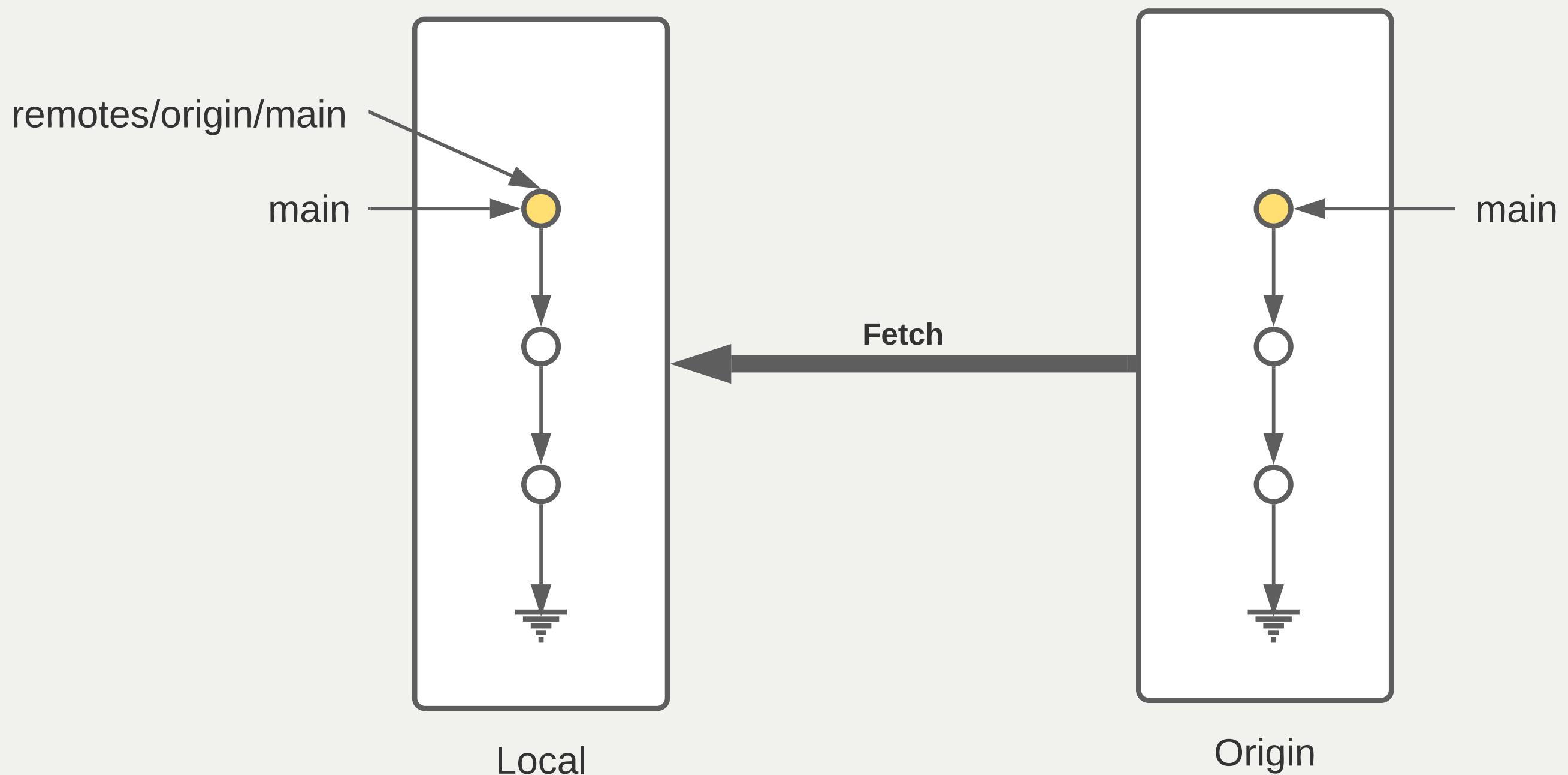
Copy

Vu que votre branche main n'a pas divergé (== partage le même historique) de la branche distante,  
git merge se passe bien et effectue 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

# 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**.

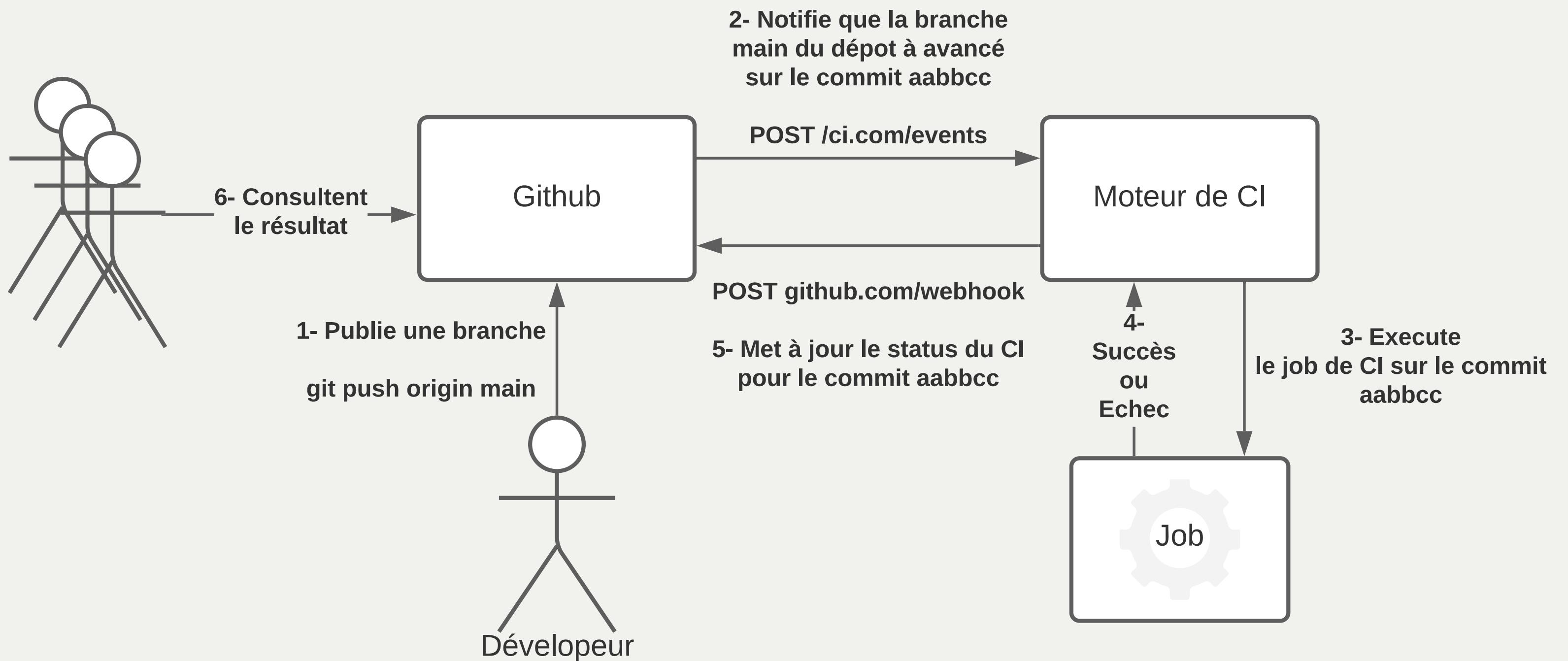
# git + Git(Hub|Lab|teal...) = superpowers !

- 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...

# Integration aux moteurs de CI/CD ?

- Pour chaque evenement important du dépôt
  - (merge, nouvelle branche poussée sur dépôt, nouvelle Pull Request)
  - Le service peut envoyer une requête HTTP pour notifier un service tiers de l'evennement.
    - Par exemple: à un moteur de CI/CD !

# Anatomie du déclenchement d'un job CI



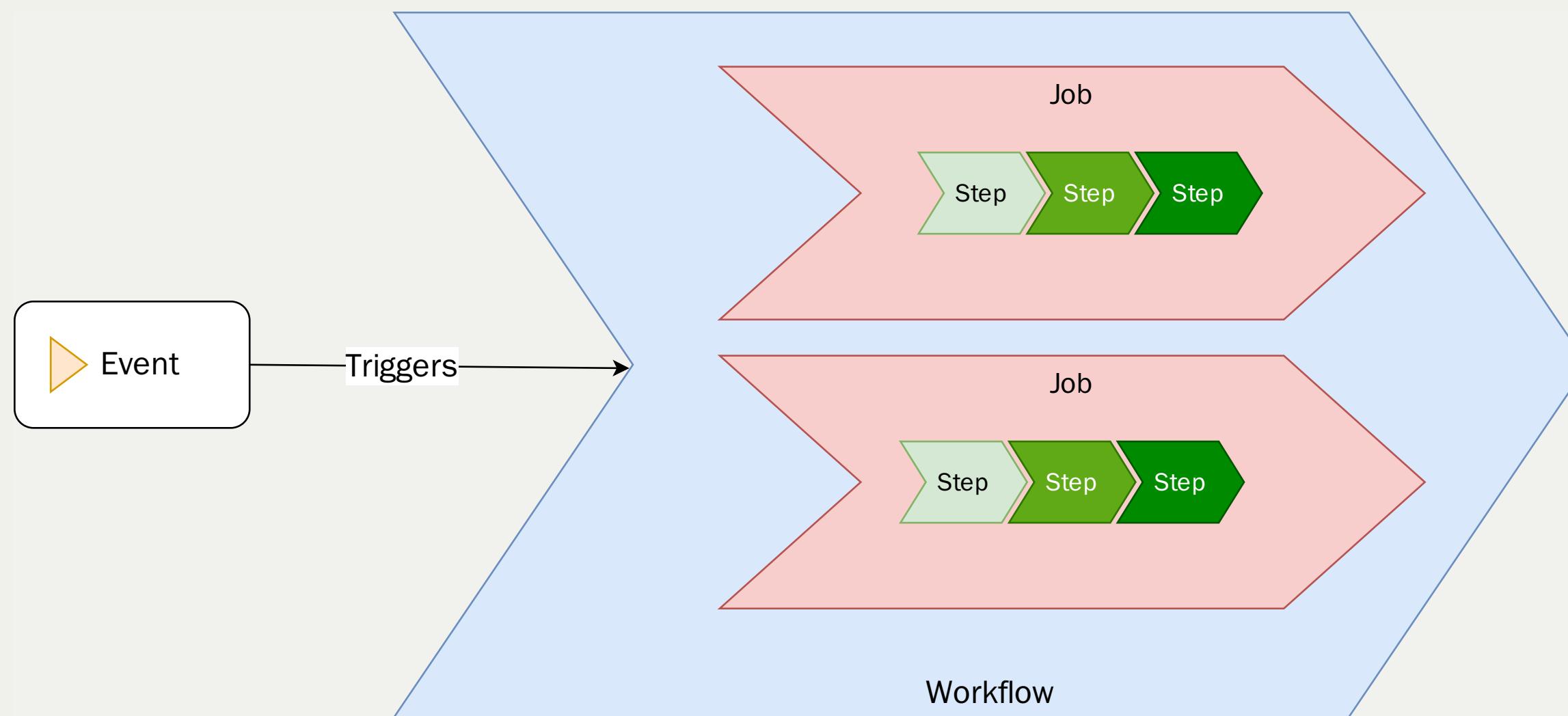
# Github Actions

# Github Actions

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

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

# Concepts de Github Actions



# Concepts de Github Actions - Step

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

# Concepts de Github Actions - Job

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

- Enchainement *séquentiel* de tâches
- Regroupement logique : "qui a un sens" (exemple : )

# 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

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

# Concepts de Github Actions - Evènement

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.
- Le workflow est exécuté pour un commit donné (Rappel : "Workflow as Code")

# Concepts de Github Actions : Exemple

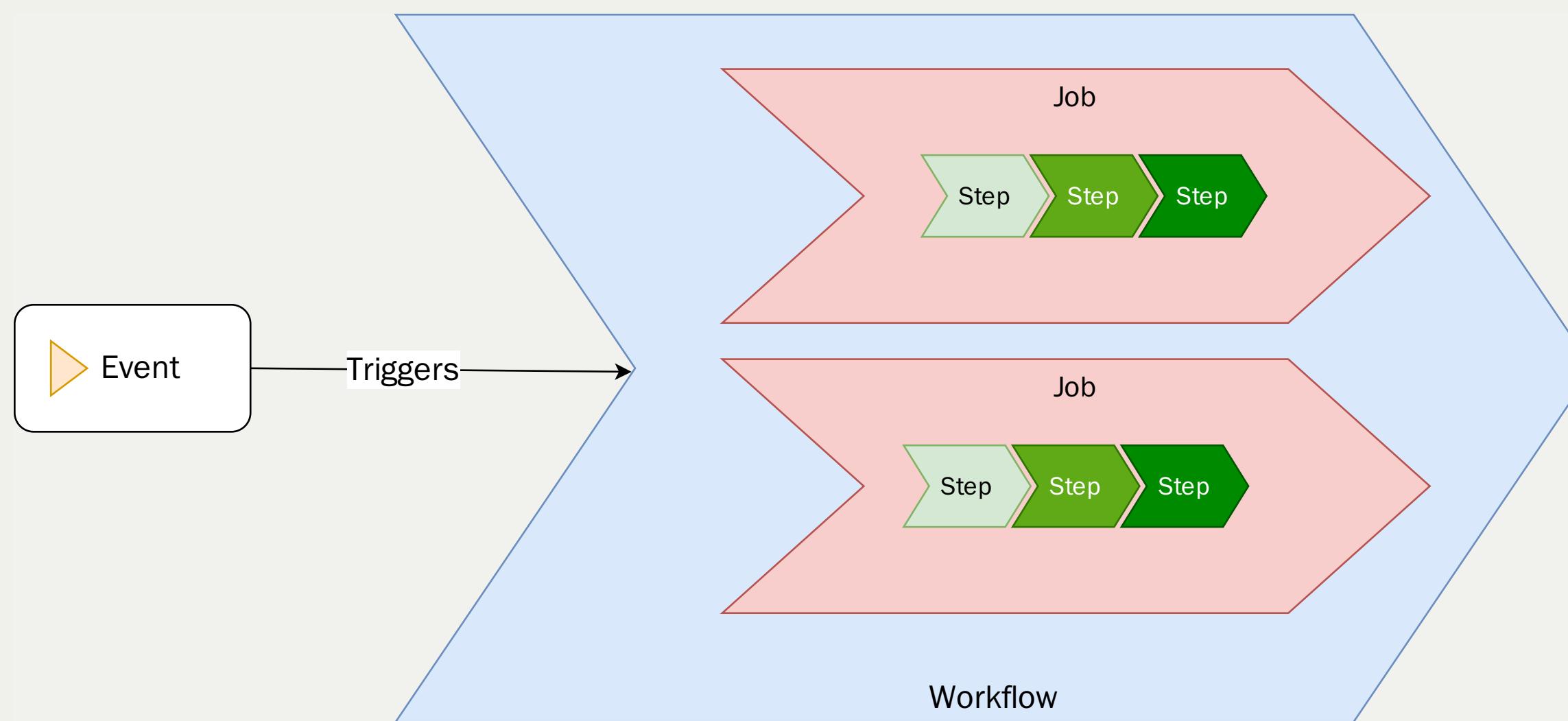
Workflow File :

```
name: Node.js CI
on: [push]
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

# Concepts de Github Actions - Récapépète



# 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

- Créez le fichier `.github/workflows/bonjour.yml` avec le contenu suivant :

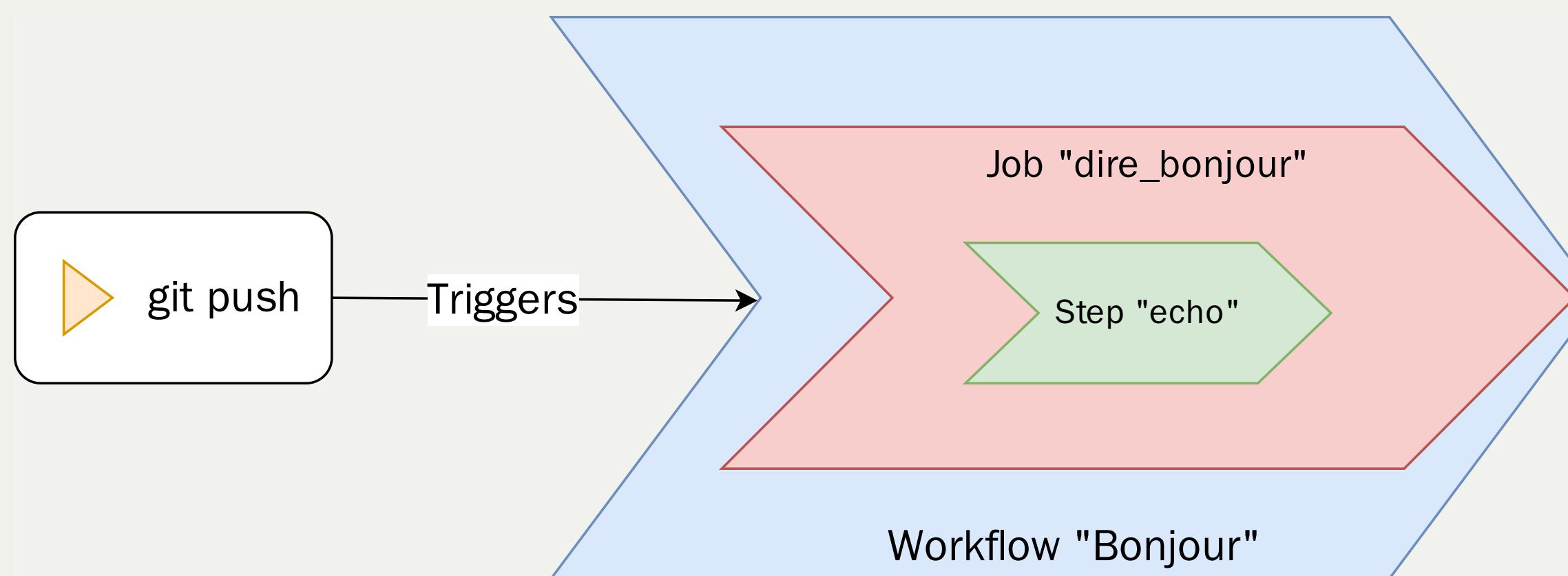
```
name: Bonjour
on: [push]
jobs:
  dire_bonjour:
    runs-on: ubuntu-18.04
    steps:
      - run: echo "Bonjour 🙋 "
```

Copy

- 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-18.04
    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-18.04
    steps:
      - uses: actions/checkout@v2 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: ls -l # Liste les fichier du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

Copy

# Exemple : Environnement d'exécution

- Notre pipeline de build dit que "la vache" doit afficher le contenu du fichier README .md
  - WAT (Non, nous ne sommes pas fous) ?
- Essayez la commande `cat README.md | cowsay` dans GitPod
  - Essayez de mettre à jour le workflow pour faire la même chose dans GitHub Actions
  - SPOILER: ✘ (la commande `cowsay` n'est pas disponible dans le runner GitHub Actions)

# Exercice : Environnement d'exécution

- **But :** On souhaite utiliser une commande spécifique durant notre job
- Deux types de solutions existent, chacune avec ses inconvénients :
  - Installer les outils manquants en préambule de chaque job (✗ lent ✓ facile )
  - Utiliser Docker pour fabriquer une action Github (✗ complexe ✓ portable)
- C'est à vous :
  - Cherchez comment installer `cowsay` dans Ubuntu 18.04
  - Appliquer cette solution dans votre job afin de le "réparer" et de voir la vache dans GitHub Actions.

# Solution : Environnement d'exécution

```
name: Bonjour
on: [push]
jobs:
  dire_bonjour:
    runs-on: ubuntu-18.04
    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

# Projet 1.0

Menu de la Cantina de Tatooine



# Netlify

Fichier `index.html` cherche environnement de production pour relation longue durée

# Rappel

- On a vu comment générer du HTML depuis un format Asciidoctor ✓
- Question: Quel est le livrable final ? Une archive tarball, un fichier `index.html`, un dossier ?

⇒ Dépends de l'environnement de production utilisé

# Production pour du HTML ?

- **But** : héberger un site web de type statique (HTML+CSS etc.)
- Il faut un service qui permet :
  - (MUST) De servir des fichiers statiques via HTTP
  - (NICE) Activer le HTTPS avec des certificats et des noms de domaines

# Say Hello to Netlify



**netlify**

<https://www.netlify.com/>

# Netlify : Kézako ?

- Platforme hébergée dans le cloud
- Offres de services gratuites et payantes en fonction des besoins
  - L'offre gratuite est suffisante pour notre besoin
- Concept de git push → netlify build → netlify deploy
  - Déjà vu ?

# Préparation : CLI Netlify

- Authentifiez-vous sur Netlify (bouton "Log In" puis "GitHub")
  - ⚠ La 1ère fois vous devrez autoriser Netlify à accéder à votre compte GitHub
- Dans l'environnement GitPod, retournez dans /workspace/exercice-makefile
- Initialisez l'environnement pour utiliser la CLI Netlify :
  - `netlify login` ([Documentation](#))
    - Si besoin, choisissez "Ouvrir dans une nouvelle fenêtre"
  - `netlify status` ([Documentation](#))
  - `netlify sites:list` ([Documentation](#))

# Créer un site avec la CLI Netlify

- Créez un nouveau "site" en suivant les étapes suivantes avec la commande `netlify site`:  
`<xxx>`:
  - `netlify sites:list` ([Documentation](#))
  - `netlify sites:create` ([Documentation](#))
    - Laissez l'option "Site Name" à vide pour obtenir un nom aléatoire
- Configurez les variables d'environnements pour Netlify:
  - `export NETLIFY_AUTH_TOKEN=XXX` (récupérez XXX dans le fichier `~/.netlify/config.json`)
  - `export NETLIFY_SITE_ID=YYY` (récupérez YYY à l'aide de la commande `netlify sites:list`)

# Déployez un site avec la CLI Netlify

- Déployez le dossier courant (`./`) contenant au moins un fichier `index.html` dans ce nouveau site
  - `netlify deploy --dir=./dist/` (Documentation)
  - Prévisualisez le site "brouillon" (🇬🇧 "Draft")
  - `netlify deploy --prod --dir=./dist/` (Documentation)
- Testez le processus complet:
  - Modifiez le contenu de `index.adoc`
  - Regénérez le HTML avec la commande `make all`
  - Re-déployez le site sur Netlify et vérifiez que votre changement est présent

# Github Actions vs. Netlify

- Netlify est une forme de CI/CD spécialisé, et Github Actions peut aussi pousser dans Netlify (CLI, etc.)
- Lequel choisir ?
  - La question elle est vite répondue jeune étudiant-entrepreneur : Github Actions
  - Pourquoi ? Pour rester homogènes et bénéficier des outils de GitHub
- Github Actions Marketplace : <https://github.com/marketplace>
  - Actions Netlify officielle ? ↗ <https://github.com/netlify/actions/tree/master/cli>
  - Alternative (non officielle) <https://github.com/marketplace/actions/netlify-actions>

# Projet 1.0 : Consignes

⇒ ⏱ Il est temps pour vous de tout assembler

- Nous sommes en 2020 et vous tenez la Cantina de Tatooine: COVID, commandes par téléphone, etc. 😷 🚲
  - Ou le Chaudron Baveur du Chemin de Traverse, le Poney Fringuant à Bree, ou même Paul Bocuse si vous ne vous sentez pas l'âme imaginaire !
- Votre projet c'est de délivrer site web contenant le menu de la Cantina
- Le code source sera hébergé au format Asciidoctor sur un dépôt GitHub
- Le site web de votre Cantina sera hébergé sur... 🚜 ...  Netlify (surprise!)
- Votre mission si vous l'acceptez : écrire un workflow github qui met en production à chaque fois qu'un changement est poussé sur main. 🚀

# Travailler en équipe ?

# Limites de bosser seul

- Capacité finie de travail
- Victime de propres biais
- On ne sait pas tout, tout le temps ?



# 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 ?

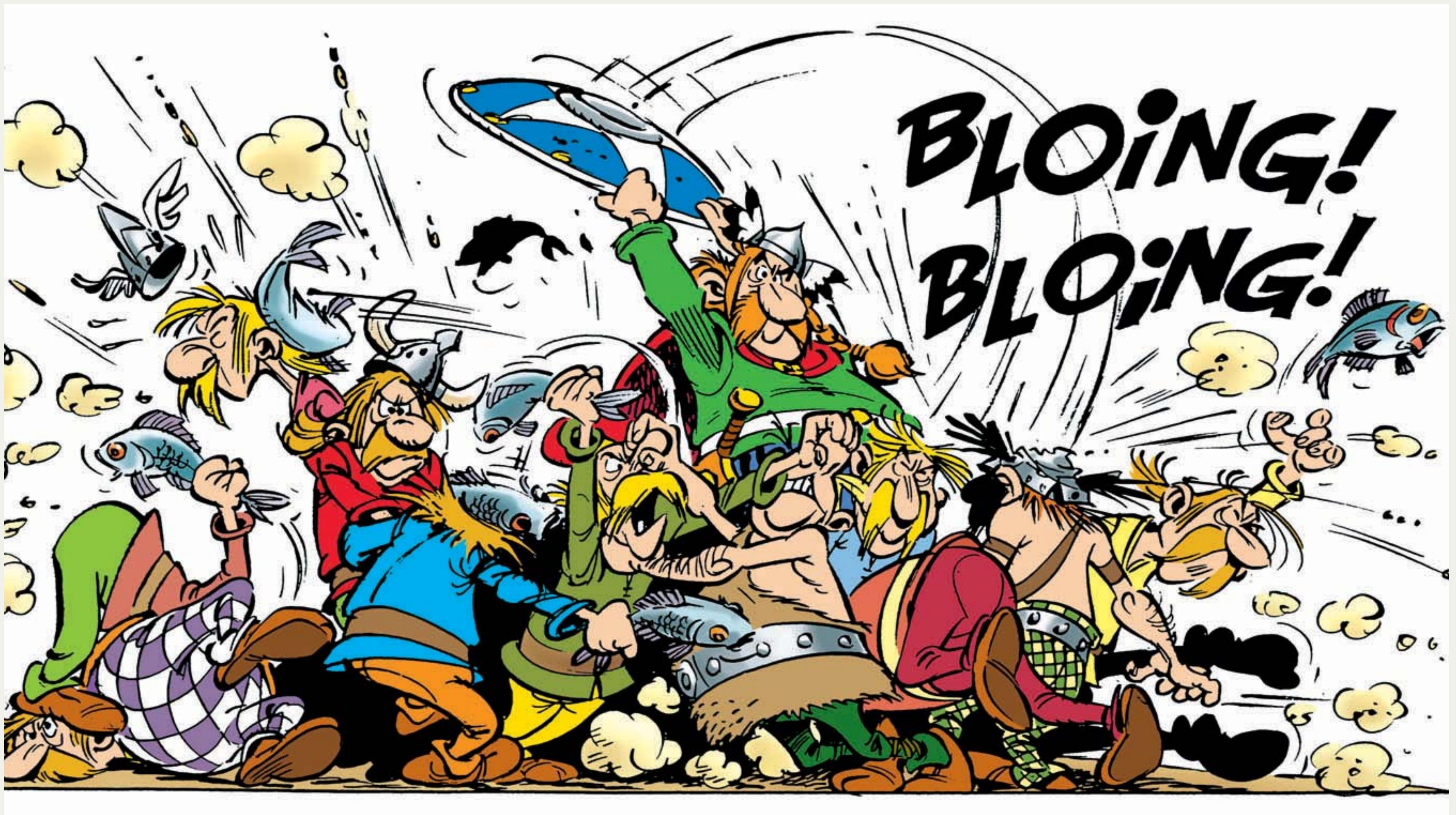
**Collaborer c'est pas évident, mais il existe des outils et des méthodes pour vous aider.**

Cela reste des outils, ça ne résoud 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 !



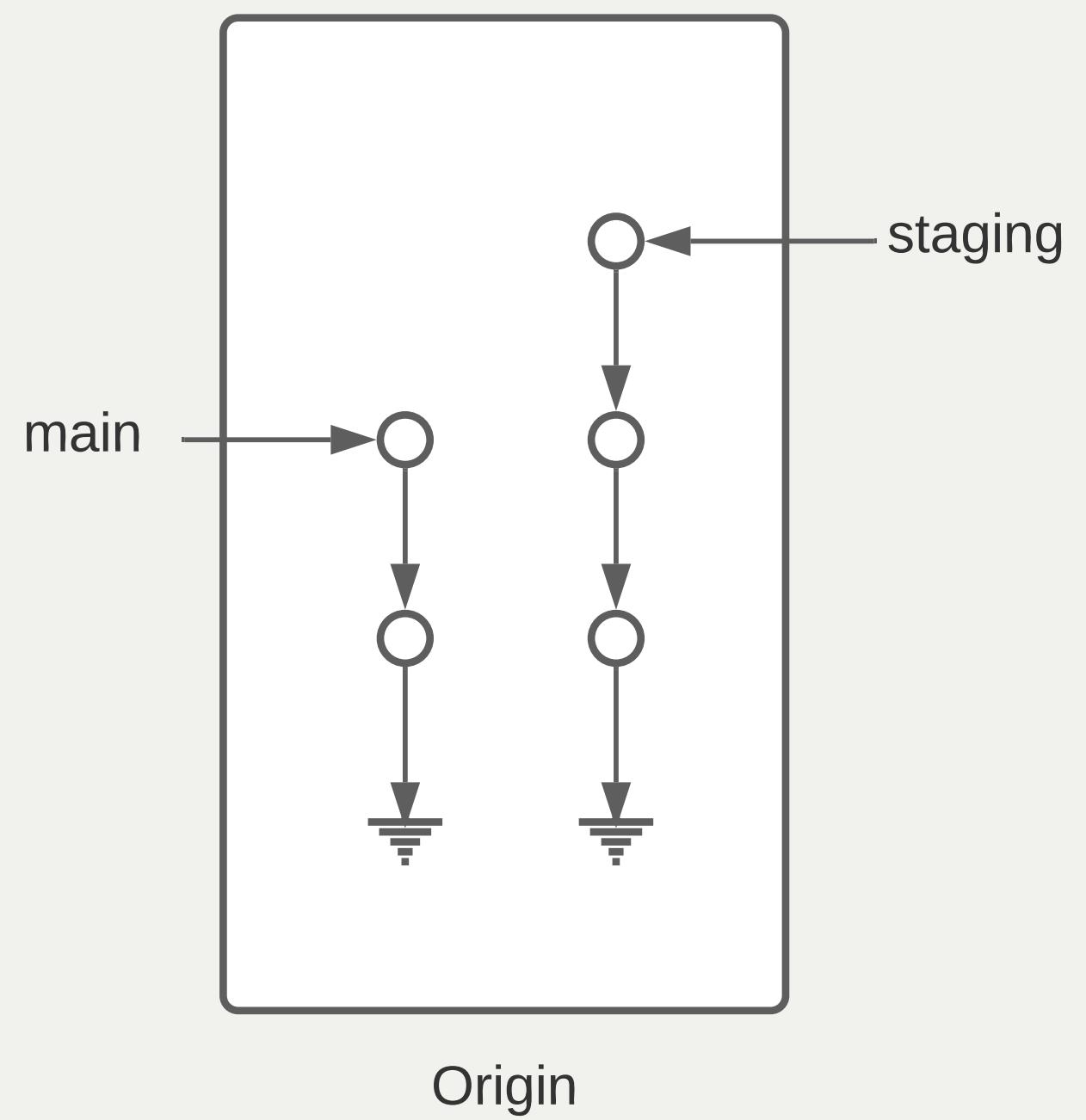
# Quelques règles pour éviter ça !

# Disclaimer

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

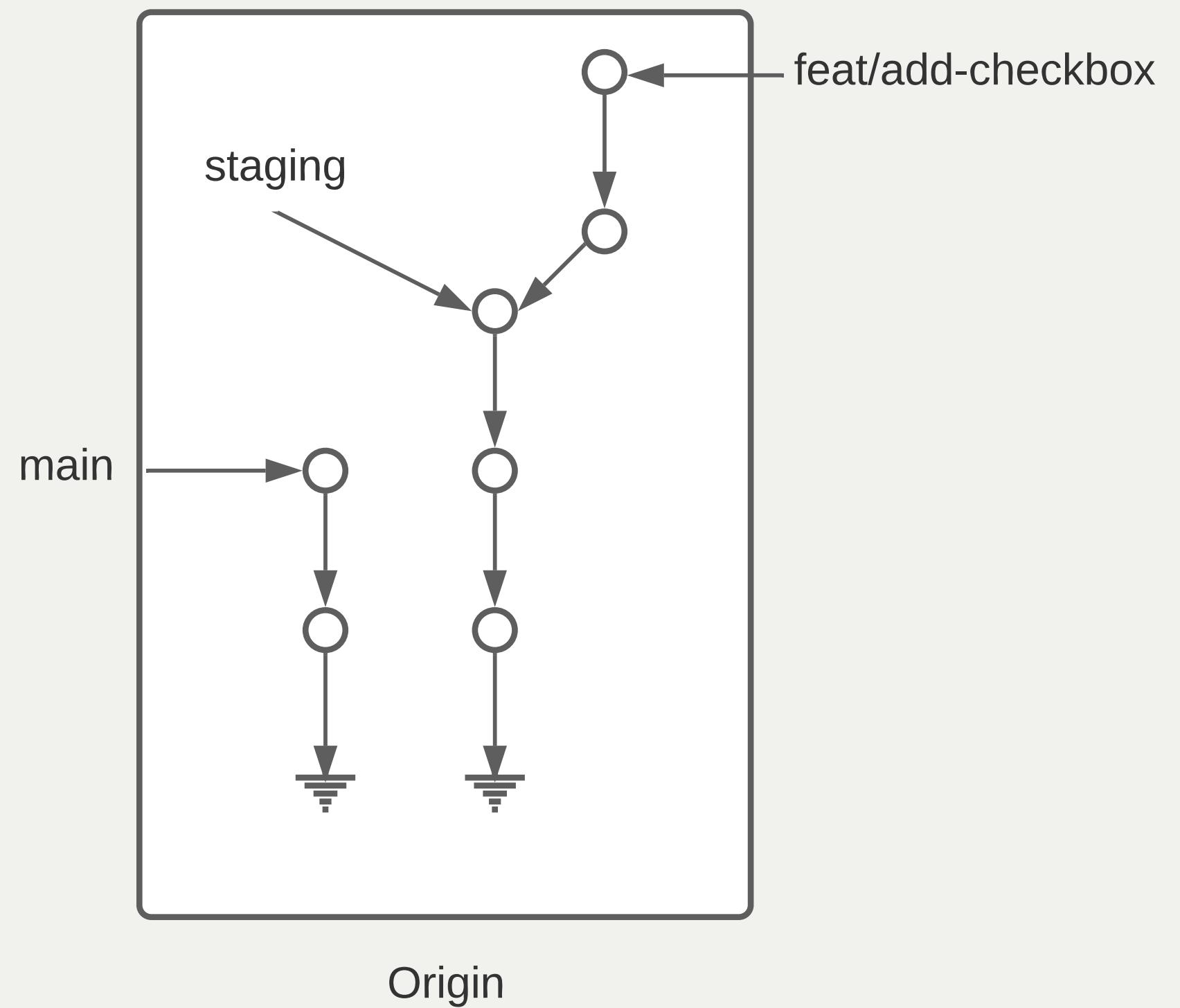
# Gestion des branches

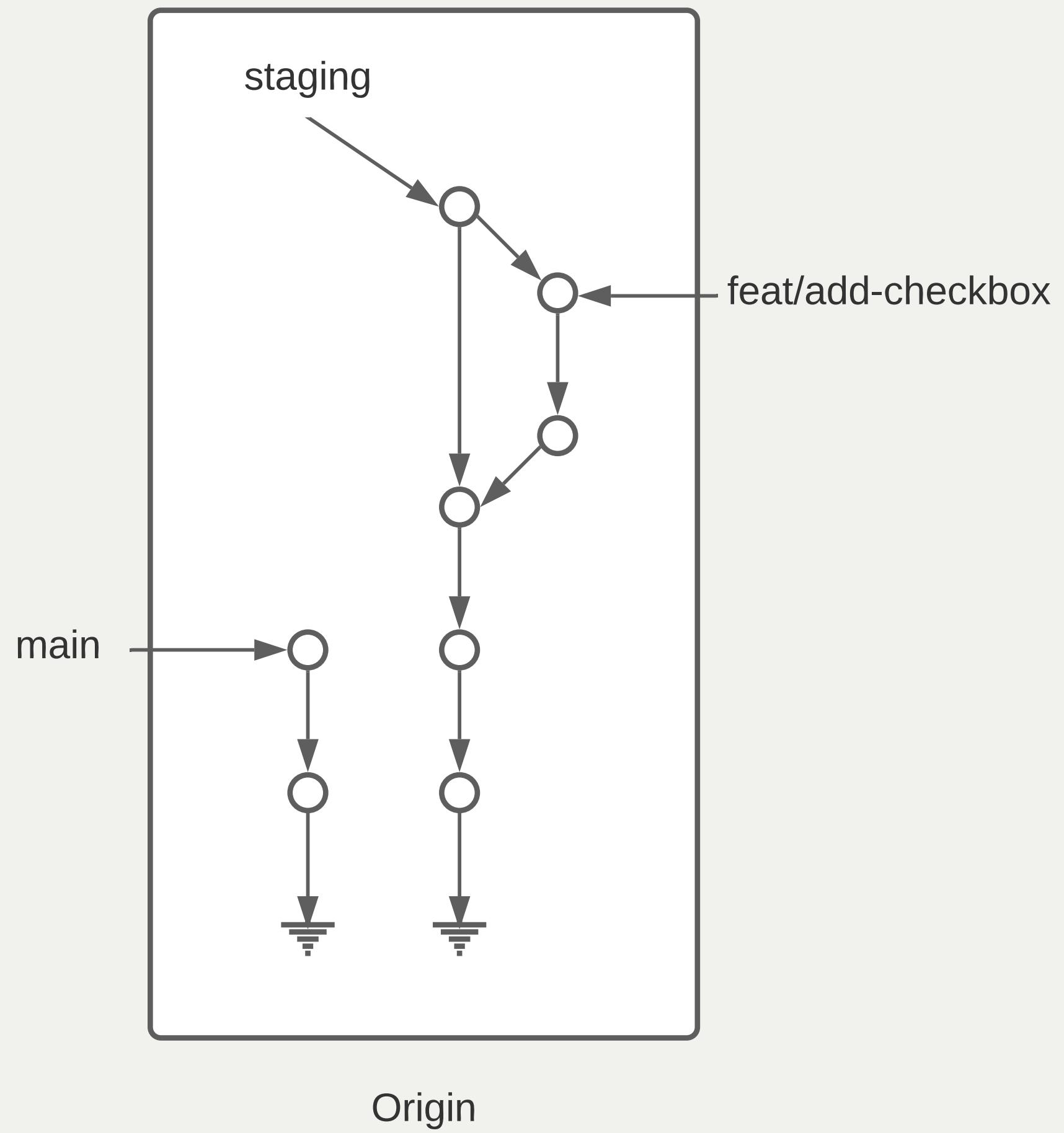
- Les "versions" du logiciel sont maintenues sur des branches principales (main, staging)
- Ces branches reflètent l'état du logiciel
  - **main**: version actuelle en production
  - **staging**: prochaine version



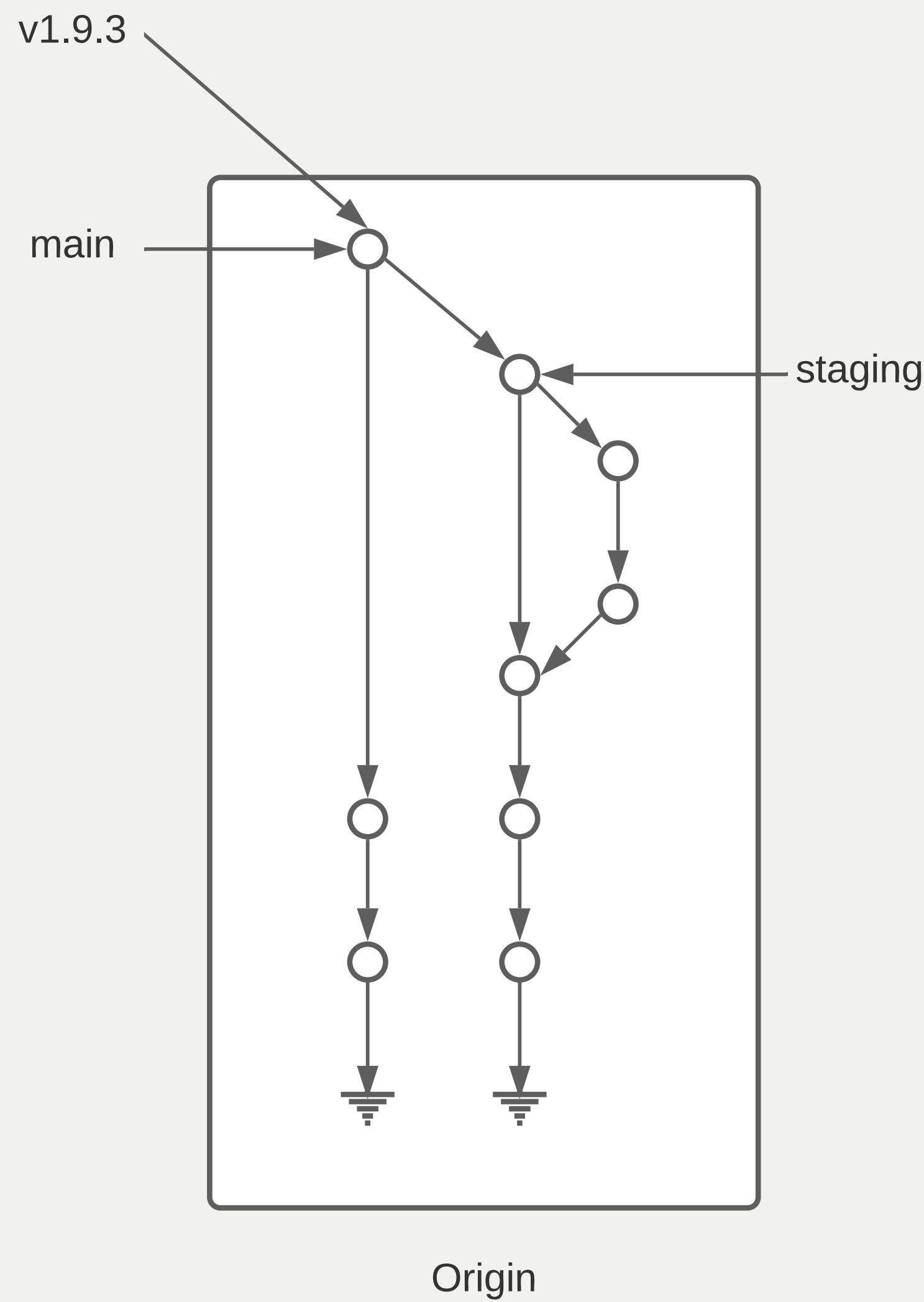
# Gestion des branches

- 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





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



# Gestion des remotes

La grande question: 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!

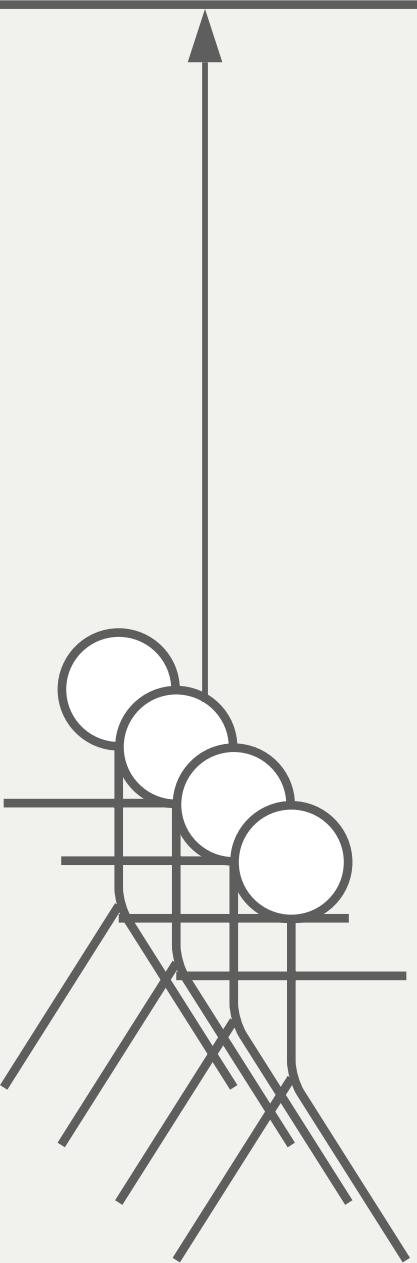
# Un remote pour les gouverner tous

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

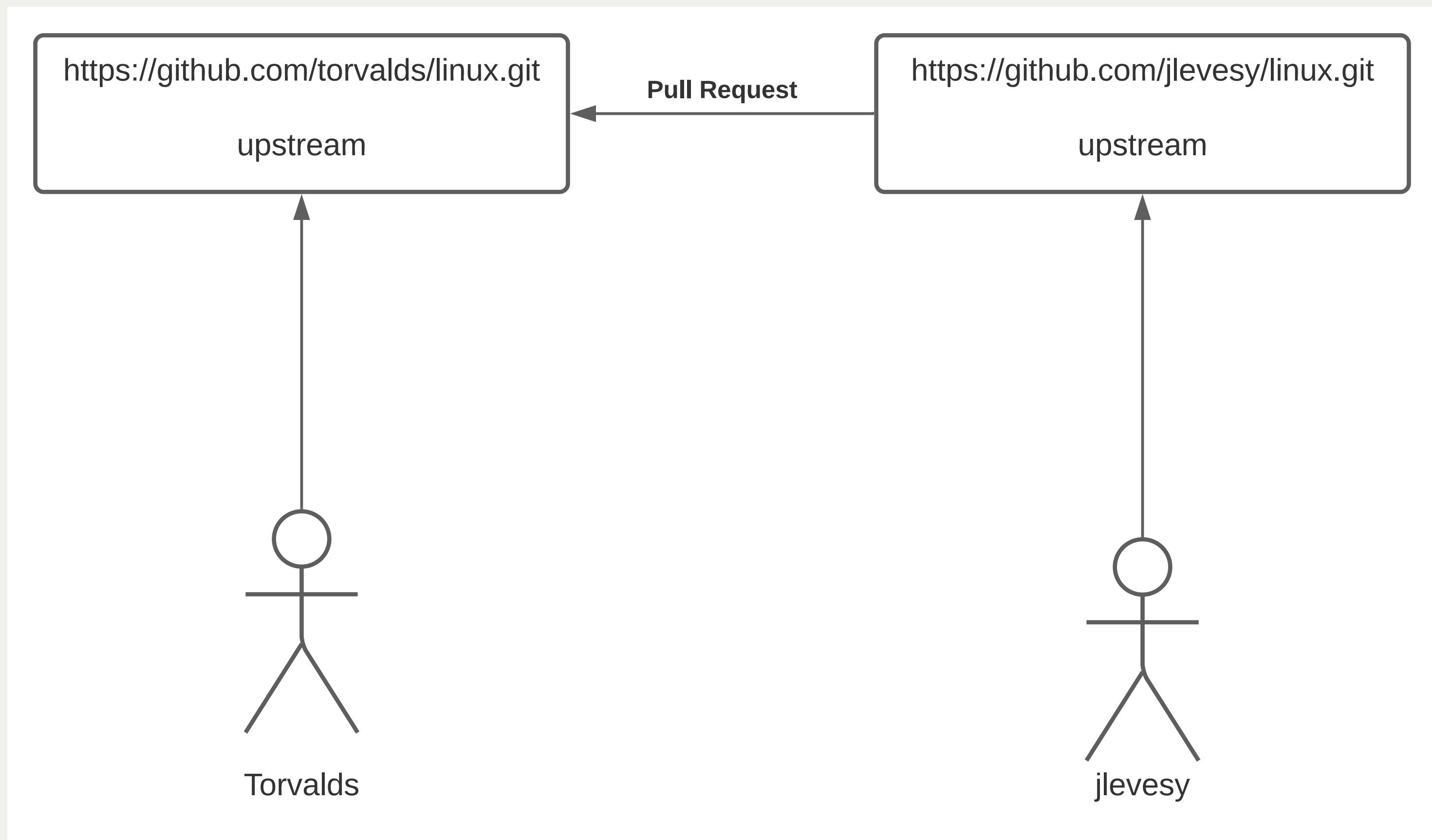
<https://github.com/torvalds/linux.git>

upstream



# Chacun son propre remote

- La motivation est 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 !



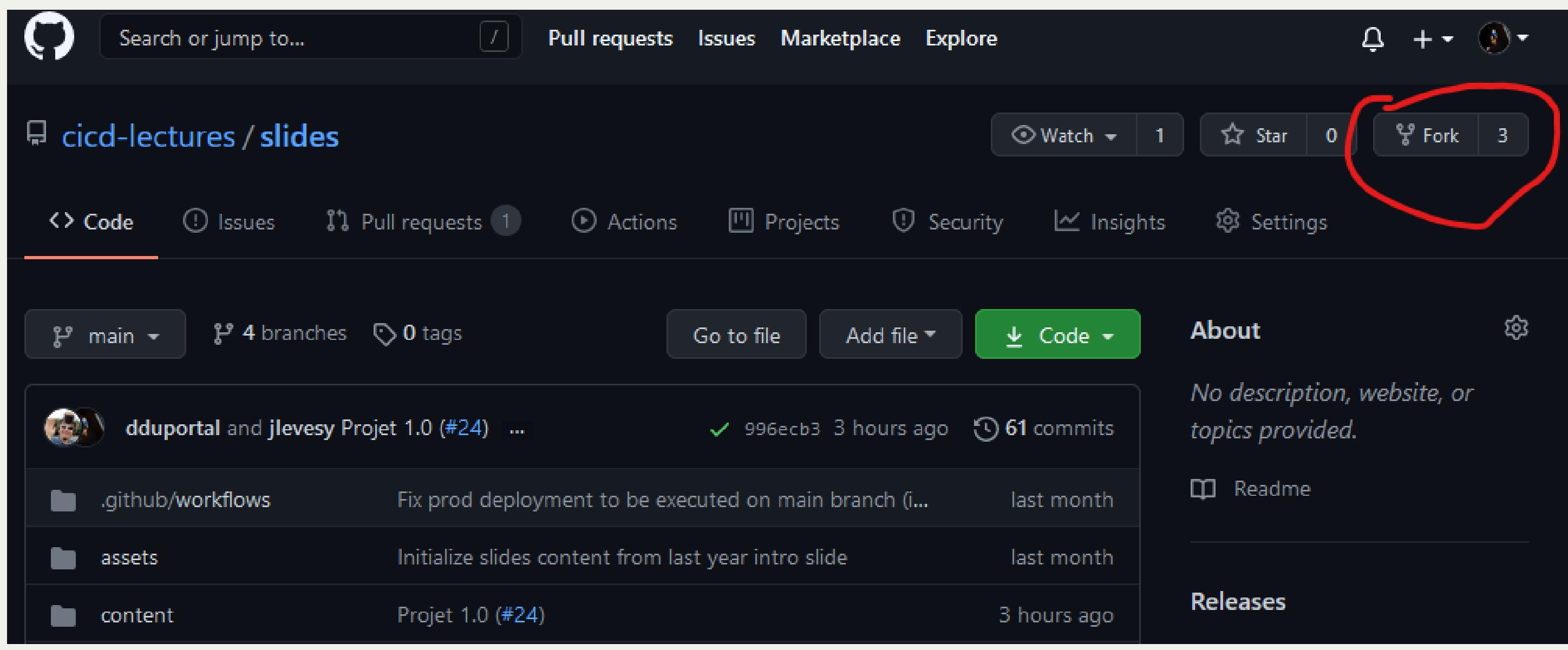
# Forks ! Forks everywhere !

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 vers un dépôt principal s'appelle une **Pull Request (PR)**.

# Exercice: Créez un fork

- Nous allons vous faire forker les dépôts créés dans le chapitre GitHub.
- 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.



# 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.

# Anatomie d'une Pull Request sur GitHub

- **Branche source:** La branche portant le changement
- **Branche cible:** La branche dans lequel le changement va être mergé.
- **Titre:** décrit de façon concise le changement apporté
- **Description:** décrit de façon détaillée le changement. Doit donner toutes les "clés de lecture" de la PR à un relecteur
- **Labels:** meta informations permettant de suivre le type de la PR (bugfix, feature?)
- **Historique de commit:** Lors d'une pull request, une attention particulière doit être portée aux commits (bien nommés, atomiques), c'est un outil aidant à la relecture!

# Exercice: faites votre changement sur votre fork

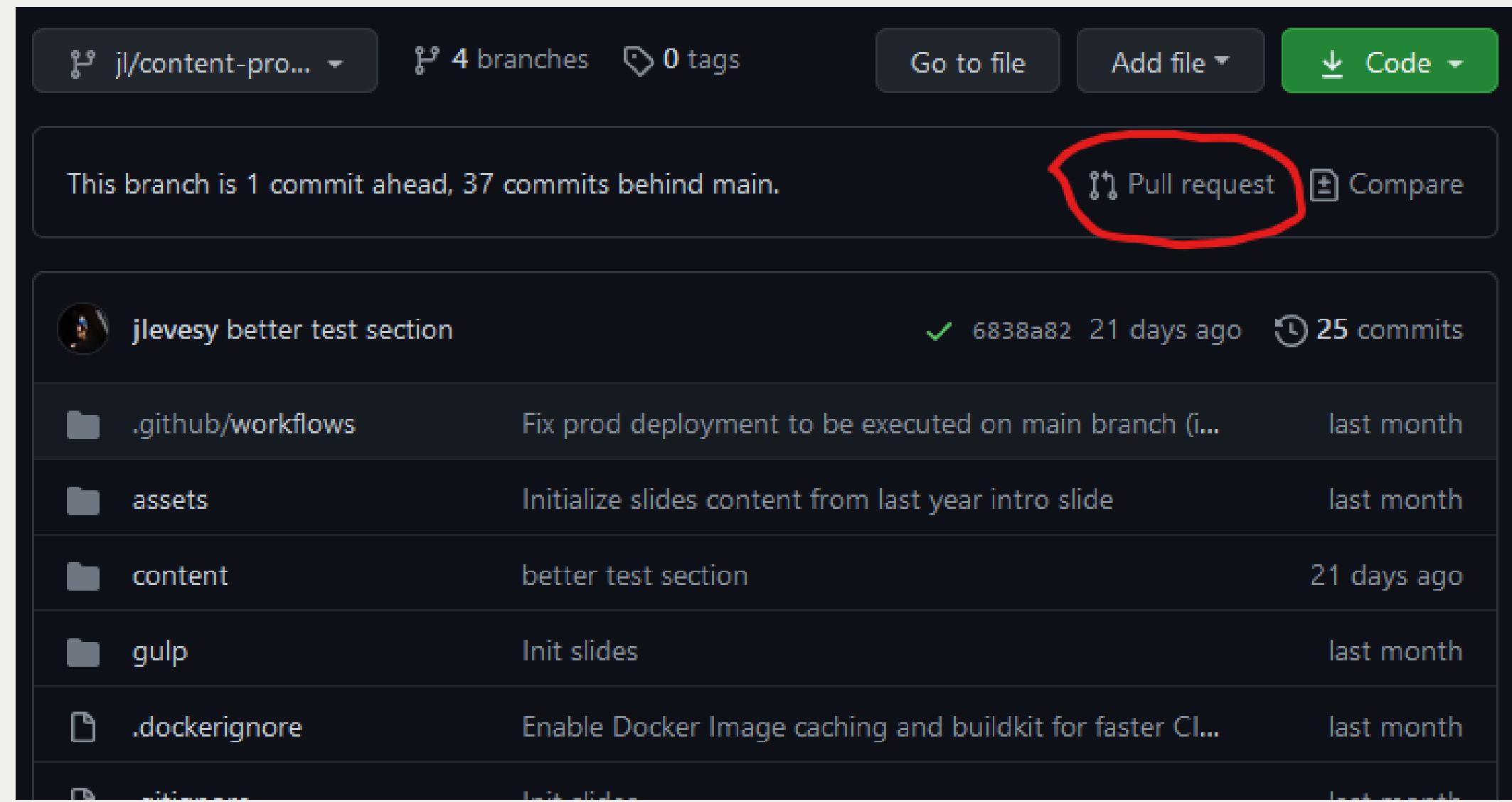
Accédez à l'environnement GitPod, puis depuis le terminal jouez les commandes suivantes:

```
cd /workspace/  
  
# Clonez votre fork  
git clone <url_de_votre_fork>  
  
# Créez votre feature branch  
git checkout -b <nom-de-votre-feature-branch>  
  
# Changez le readme ou ajoutez un nouveau fichier.  
# (bonus si c'est rigolo :p)  
# Et comitez le ;)  
  
# Publiez votre changement sur votre remote "forké"  
git push origin <nom-de-votre-feature-branch>
```

Copy

# Exercice: Ouvrez votre PR

- 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.



# 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 elle vous convient, mergez la pull request dans votre dépôt.

# 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
- Execution des tests automatisés du projet (unit, integration)
- Déploiement du projet dans un environnement de test (coucou Gitpod.io ou Netlify !)

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

## 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.

# Projet 1.1 : Contribuer au Menu

Open sourcer le projet 1.0 !

# Règles du jeu

- Chaque contributeur externe pourra proposer des changements au menu
- Le dépôt du menu doit être en lecture publique.
- Seul le mainteneur du menu (vous!) peut écrire dessus.
- L'objectif:
  - Faire en sorte que pour chaque PR ouverte sur le dépôt le contenu du menu soit validé par un job de CI
  - Ce workflow va valider que le HTML généré suite au changement est "correct"

# Challenge : Tester du HTML

- Ouverture du menu à la contribution externe : comment tester le HTML ?
  - 🔎 Il faut valider/tester les modifications
  - 🌐 Nous allons tester si les liens HTTP sont valides et pointent vers des pages existantes
- Essayez avec la commande linkchecker :

```
make clean main.html && linkchecker --check-extern ./main.html
```

Copy

- Modifiez le contenu pour tester le cas d'un mauvais lien HTTP et ré-essayez

# Pré-requis : Solution Projet 1.0

Voici une proposition de solution (ce n'est donc pas la seule solution possible !) pour le Projet 1.0 :

```
name: Cantina
on: [push]
jobs:
  cantina-menu:
    runs-on: ubuntu-18.04
    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 asciidoctor # Installation des outils requis
      - run: make all # Génération du dossier livrable `./dist/`
      - uses: netlify/actions/cli@master # Déploiement sur Netlify
        with:
          args: deploy --prod --dir=./dist/
    env:
      NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }} # A définir dans https://github.com/<votre dépôt github>/settings
      NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN }} # A définir dans https://github.com/<votre dépôt github>/se
```

Copy

# Projet 1.1 : Consigne

Faire en sorte que le CI execute un job différent en fonction du type d'événement:

- Si c'est un commit d'une PR: build + test uniquement
- Si un nouveau commit est poussé sur main: build + test + deploy
- Plusieurs options:
  - Un seul workflow avec un **step conditionnel**
  - Un **workflow spécifique**

# Projet 1.1.1 : Continuous delivery vs Continous deployment

- Il n'est pas toujours adapté de déployer automatiquement la branche main
- Vous décidez donc de changer votre job pour qu'il ne déploie que sur un **tag**

# Versions

# Pourquoi faire des versions ?

- Un changement visible d'un logiciel peut nécessiter une adaptation de ses utilisateurs
- ... or dans certains cas l'adaptation n'est pas automatique !

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 a "changé le contrat" de son API d'une façon non rétrocompatible avec votre l'existant.
  - Cela s'appelle un  **Breaking Change**

# Comment éviter cela ?

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

Pour effectuer cela, il est nécessaire de rendre manipulable facilement la notion de version!

# Bonjour versions !

- Une version cristallise un contrat respecté par votre application.
- C'est un jalon dans l'historique de cette dernière.

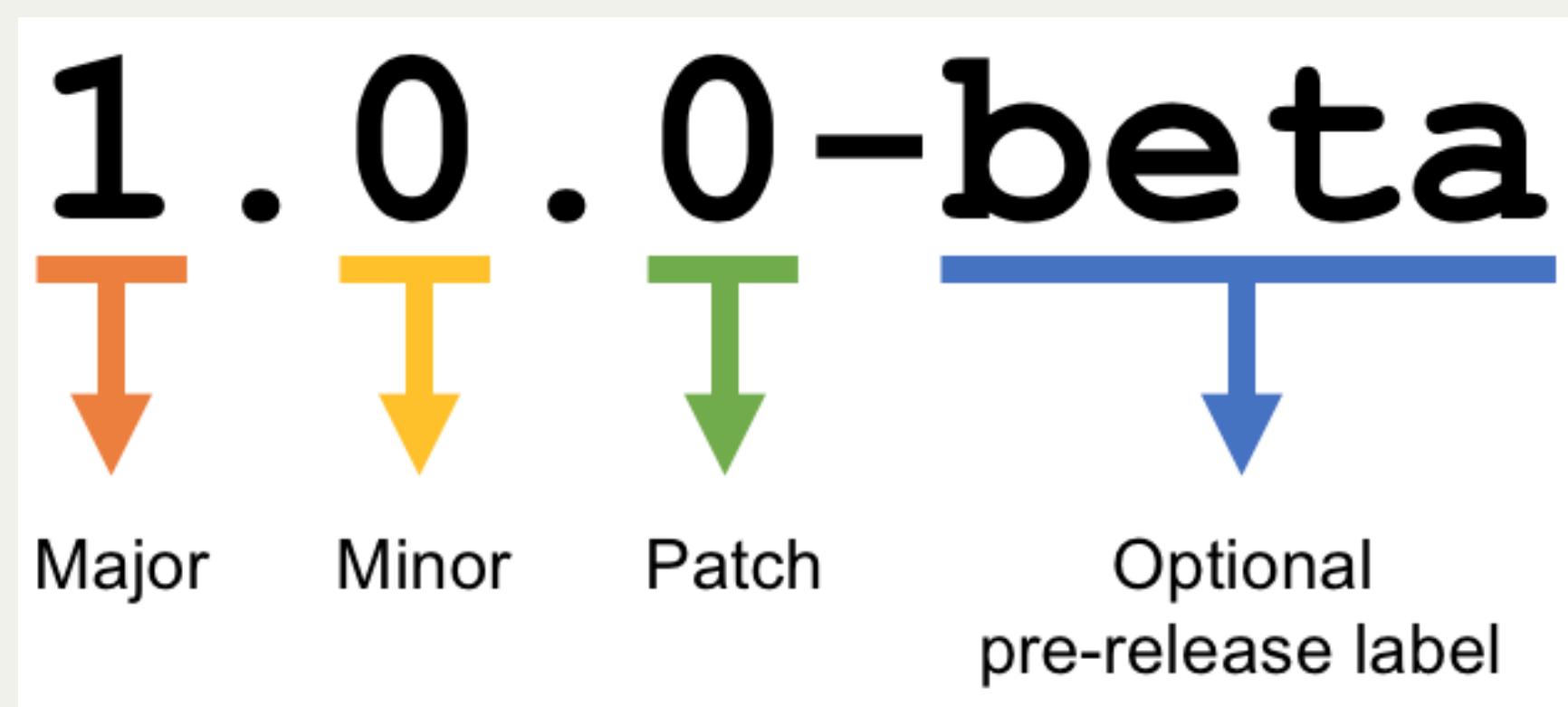
# Quoi versionner ?

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.

```
# Créer un tag.  
git tag -a v1.4.3 -m "Release version v1.4.3"  
  
# Publier un tag sur le remote origin.  
git push origin v1.4.3
```

Copy

# Maven : Niveau 2

# Maven ?

Pourquoi est-ce qu'on s'embête avec des outils comme Maven ?

Make c'est bien suffisant non ?

# Do It Ourselves or Reinvent the Wheel ?

**Problème :** Doit-on recoder tous ses outils ou réutiliser des choses existantes ?



**Réponse :** ça dépend donc ce sera à vous de juger et de ré-évaluer

# 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

# Pourquoi Maven ?

- On fait du Java...
  - Alternatives en Java : Gradle, Bazel, Ant
- Bon exemple d'application car complet (cycle de vie, configuration, dépendances)
- Plutôt mature (1ère release : 2004)

# Maven : pom.xml

- Maven a besoin d'un fichier pom.xml à la racine de votre projet
- XML : language de type "markup", avec un **schéma**, donc strict
- "POM" signifie "Project Object Model"
- Concept de "Convention au lieu de configuration" pour limiter la complexité

```
<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>

  <!-- Contenu du fichier pom.xml -->

</project>
```

Copy

# Maven : Identité d'un projet

Maven identifie un projet par l'**•**es artefact**•**s générée**•**s , en utilisant les 3 éléments **obligatoires** suivants :

- **groupId** : Identifiant unique de votre projet suivant les règles Java de nommage de paquets
- **artifactId** : Nom de l'artefact généré par votre projet
- **version** : Version de l'artefact, qui **devrait** respecter le semantic versionning.
  - Peut être suffixé par –SNAPSHOT pour indiquer une version non releases.

```
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0-SNAPSHOT</version>
```

Copy

# Exercice : Maven From Scratch

⇒ C'est à vous dans l'environnement GitPod

- Créez un projet vide pour Maven :

```
mkdir -p /workspace/mvn-level2/src/main/java && cd /workspace/mvn-level2
```

Copy

- A partir des 2 slides précédentes, créez un fichier pom.xml avec la balise project qui définit les schémas, contenant 4 autres balises : modelVersion, groupId, artifactId et version
- Créez 1 fichier "Hello.java" dans src/main/java/ avec le contenu ci-dessous :

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

Copy

- Essayez de compilez le projet avec mvn compile (SPOILER: ✘)

# Définir la plateforme d'exécution

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 :
  - ✗ **Source** option 5 is no longer supported. Use 7 or later
  - ✗ **Target** option 5 is no longer supported. Use 7 or later
  - ⚠ File encoding has not been set, using platform encoding ANSI\_X3.4-1968, 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 ? Ecrivez 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 ?
- Il faut définir la cible d'exécution (e.g. la **production**) du programme

# 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 utilisée 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 : Définir la plateforme d'exécution

**But :** la commande mvn compile doit fonctionner sans erreur, et produire un fichier Hello.class dans ./target/\*\*

1. Modifiez le fichier pom.xml pour ajouter un bloc <properties> et définissez la valeur de la propriété project.build.sourceEncoding à UTF-8 (résolution du warning).
2. Utilisez la documentation du Maven Compile Plugin pour résoudre les 2 erreurs de compilation
  -  Utilisez la version majeure de java -version

# Solution : Définir la plateforme d'exécution

```
<properties>
    <maven.compiler.source>15</maven.compiler.source>
    <maven.compiler.target>15</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Copy

# Exécuter l'Application

██ Succès !

mvn compile a produit le fichier ./target/classes/Hello.class

Exécutons notre programme avec la commande java:

```
# "-cp" == "classpath" (Chemin vers les classes Java "compilées")
java -cp ./target/classes/ Hello
# Argument "Hello" == classe qui contient la méthode statique "main"
```

Copy

# 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>
- **Remote** : un dépôt de votre organisation, similaires à un remote GitHub, hébergé par vos soins
- **Local** : un dossier sur la machine où la commande mvn est exécuté, généralement dans  
`\$ { HOME } / .m2`

# Dépendances Maven

Pour spécifier les dépendances :

- 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

- Revenons aux exercices à base de tests : nous avons utilisé le framework Spring
- **Idée** : c'est un framework pour ne pas avoir à tout ré-écrire, exécuté lorsque l'application est en fonctionnement : c'est donc une *dépendance* de notre application.

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

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.3.3.RELEASE</version>
  </dependency>
</dependencies>
```

Copy

# Exercice avec les dépendances Spring

⇒ C'est à vous. Ajoutez le bloc précédent dans votre pom.xml

- Exécutez la commande mvn clean compile
- Explorez le contenu du dossier \$HOME/.m2 (écriture équivalente à ~/ .m2)
  - En particulier :

```
ls -l ~/.m2/repository/org/springframework/boot/spring-boot-starter-web/2.3.3.RELEASE
```

Copy

et

```
ls -l ~/.m2/repository/org/apache/maven/plugins/
```

Copy

- Supprimez le dossier ~/ .m2 / et relancez la commande mvn clean compile

# Solution avec les dépendances Spring

- Le dépôt local .m2 :
  - Agit comme un "cache" local contenant dépendances et plugins
  - Respecte la structure des groupId, artifactId et version
- Commande mvn install :
  - Exécute les étapes package et verify
  - Puis copie le résultat de package dans le dossier .m2
  - Essayez mvn install puis vérifiez le contenu de ~/.m2/repository/<groupId> en format <dossiers>/<artifactId>/<version>

```
ls -l ~/.m2/repository/com/mycompany/app/my-app/1.0-SNAPSHOT/
```

Copy

# Convention Over Configuration

- Maven fonctionne à base de "convention": lorsque nous avons corrigé les erreurs de compilation, le plugin Maven Compiler **s'attendait** à avoir des propriété définies comme défini dans la documentation.
- On peut également "configurer" très finement Maven à l'aide des balises XML du pom.xml

# Exercice : Changer le nom de l'artefact final

- **But:** Produire un artefact JAR dont le nom est constant
- Toujours dans l'environnement GitPod, exécutez la commande mvn package
- 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 hello.jar**.

# Solution : Changer le nom de l'artefact final

```
<build>
  <finalName>hello</finalName>
</build>
```

Copy

# Maven Plugins

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

- "**Build**" : Implémente une action durant les phase de "build" (clean, compile, test, etc.), et est configuré dans la balise <build>
- "**Reporting**" Implémente une action durant la phase de génération de "site", et est configuré dans la balise <reporting> (à votre grande surprise)

# Exercice : Maven JAR Plugin

- **But:** Produire l'artefact JAR dans un dossier nommé `dist` à côté du `pom.xml` et de `target` /
- La génération du JAR est déclenchée lors de l'appel à `mvn package`, il nous faut une documentation !
  - Est-ce qu'il y a un plugin `package` dans la page de la liste des plugins Maven ?
  - A vous de chercher pour trouver la documentation du plugin et d'y trouver le bon réglage permettant de changer le dossier d'"output"

# Solution : Maven JAR Plugin

- <https://maven.apache.org/plugins/>
  - <https://maven.apache.org/plugins/maven-jar-plugin/>
    - <https://maven.apache.org/plugins/maven-jar-plugin/jar-mojo.html>

```
<build>
  <!-- ... -->
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <executions>
        <execution>
          <configuration>
            <outputDirectory>./dist/</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Copy

# Projet 2.0

# Enoncé

- Votre prédécesseur•se a écrit une application pour le site web de la Cantina, mais a soudainement démissionné pour partir éllever des serpents sur Dagobah.
- Cette application gère la banque de menus de la cantina. On peut créer et récupérer les dits menus. Un menu est composé de plats (Dishes).
- En arrivant à la Cantina, le patron vous a envoyé un lien vers une archive contenant le code source Java d'une application Spring Boot, plus ou moins bien instrumentée...
- Votre mission : Faire en sorte d'industrialiser cette application avec les connaissances acquises durant ce cours.

# Récupérer l'application

- Vous pouvez récupérer ici l'archive Tar-gzippé nommée `project-2.0.1-src.tar.gz`.
- Empreinte SHA256 :  
`ddc634a00f64a1606f3d813e15a7e66a08a87d0cec313b0dee3bb157e9e6ddd0.`

# Un peu plus de détails: VCS

- Versionner et héberger le projet dans un dépôt public GitHub ou GitLab
- Possède une organisation de branches **représentant le cycle de vie de l'application.**
  - Deux branches: main et development
  - main ne devrait avoir que des **commits de merge** (Via PR ou non ?) issus de development.
  - Chaque commit de main est "taggé" et correspond à une release du logiciel avec une **version sémantique**.
  - La branche development ne devrait avoir que des **commits de merge** issus de PRs.
- Un historique de commits à peu près propre :)

# Un peu plus de détails: PRs

- On vous conseille de travailler sur le projet uniquement par PRs (Pull Requests) qui feront office de documentation de votre travail
- Par exemple:
  - PR-1: Mise en place du job de CI
  - PR-2: Mise en place de Maven et activation de la compilation dans le job CI
  - PR-3: Ajout des tests unitaires et activation dans job le CI
  - PR-4: Ajout des tests d'intégration et activation dans le job CI
  - PR-5: Mise en place du job de CD

# Un peu plus de détails: Tests

S'assurer que la couverture de test est "satisfaisante" pour la base de code fournie :

- Des tests unitaires et des tests d'intégrations sont déjà présents mais doivent être corrigés ou complétés
  - Pas de nouveau fichier à créer
  - Les tests présents sont considérés comme suffisants pour ce projet

# Un peu plus de détails: Maven

Utilisez maven pour gérer les dépendances et implémenter le cycle de vie technique de l'application :

- `compile`: compile l'application
- `package`: crée un jar(jar) exécutable avec la commande `java -jar <fichier.jar>`
- `test`: exécute les tests unitaires
- `verify`: exécute les tests d'intégration

# Un peu plus de détails: CI et CD

Ce projet devra être associé à un moteur d'intégration continue  
(GitHub Actions, GitLab CI ou autre si vous êtes joueurs, la seule contrainte est que ce soit un SaaS gratuit et accessible publiquement)

# CI: Sur une branche de travail

- Compile l'application
- Joue les tests unitaires
- Joue les tests d'intégration
  - Bonus si récupère les rapports de tests comme artifacts du job

# CI: Sur un push dans main

- Compile l'application
- Joue les tests unitaires et intégration

# CD: Sur tag pushé

- Compile l'application
- Joue les tests unitaires et intégration
- Fait une release GitHub avec le fichier JAR de l'appication (wink wink) attaché, dont le nom correspond a la version du tag.
  - Bonus si Maven tient compte de la version tagguée ;)

# Pré-requis : Solution Projet 1.1 - Makefile

Voici une proposition de solution (ce n'est donc pas la seule solution possible !) pour le Projet 1.1 :

```
.PHONY: all
all: clean dist test

main.html:
    asciidoctor main.adoc

.PHONY: dist
dist: main.html
    mkdir -p ./dist
    cp ./main.html ./dist/index.html

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

.PHONY: test
test: main.html
    linkchecker --check-extern ./main.html
```

Copy

# Pré-requis : Solution Projet 1.1 - Workflow

```
name: Cantina
on: [push, pull_request]
jobs:
  cantina-menu:
    runs-on: ubuntu-18.04 # Required for linkchecker
    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 asciidoctor linkchecker # Installation des outils requis
      # Génération du dossier livrable `./dist/` avec test inclus
      - run: make clean
      - run: make test # Cible à définir dans le Makefile évidemment
      - run: make dist
      - uses: netlify/actions/cli@master # Déploiement sur Netlify
        if: contains(github.ref, 'main') # Seulement si la "ref" git contient "main" (d'autres solutions sont possibles)
    with:
      args: deploy --prod --dir=./dist/
    env:
      NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }} # A définir dans https://github.com/<votre dépôt github>/settings
      NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN }} # A définir dans https://github.com/<votre dépôt github>/se
```

Copy

# Critères d'évaluation

- Les critères d'évaluation sont détaillés sur cette page: Notations ENSG 2020/2021, selon les grandes catégories suivantes :
  - VCS / GitHub / GitLab : 6 points
  - Tests : 5 points
  - Maven : 4 points
  - CI/CD : 5 points

# Consignes de rendu

- Envoi de l'email pointant vers ces consigne mises à jour le 09 janvier 2021
- **Deadline du rendu:** 5 semaines à partir du jour de livraison de l'application initiale, soit le 13 février 2021
- Vous devrez nous envoyer un mail (par binôme) avec :
  - Pour chaque membre du binôme:
    - Nom et prénom
    - Email
    - Identifiant GitHub ou GitLab utilisé
  - Le lien vers votre dépôt de rendu

# Rendu des notes

- Les notes seront rendues 3 semaines après la deadline, soit pour le 6 mars 2021 au plus tard
- Contestation/relecture : vous aurez ~ 1 semaine après le rendu des notes si jamais vous n'êtes pas d'accord ou souhaitez une clarification

# Un dernier mot

- Amusez vous !
- Ne passez pas plus de 10h dessus !
- Vous êtes là pour apprendre, pas pour vous rendre malade !

# Merci !

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

Slides: <https://gifted-albattani-3bc8a1.netlify.app/>



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