

# Git

**AUKFOOD**  
Infogérance et conseil en logiciel libre

# Introduction

# Gestion de version

=

# Versioning of Code

## Avantages :

Pour chaque modification :

- L'Auteur
- La Date
- les Modifications

Travail collaboratif

## Inconvénients :

Courbe d'apprentissage

Multitude d'outils

# Gestionnaire de version

## Modèle centralisé

Toutes les données se trouvent sur un serveur central

Toute la gestion se fait avec le serveur central

Difficile d'utilisation sans connexion avec le serveur

**Inconvénient :**

Serveur = SPOF

# Gestionnaire de version

## Modèle décentralisé

Les données et la gestion s'effectuent en local sur le poste de l'utilisateur

Plus de SPOF

Les données peuvent être centralisées pour un travail à plusieurs mais il y aura toujours des copies disponibles

### Inconvénient :

La récupération d'un code récupère aussi tout l'historique

# Gestionnaire de version

Travailler à plusieurs

Création de branche

Gestion des conflits

Fusionner

Retour arrière possible

# Gestionnaire de version Avancé

Intégrations et  
Déploiement continu

Registry Docker

Gestion de projet  
(Kanban)

Wiki

Gestion de groupe  
utilisateurs

Suivi des bogues  
(issues Tracker)



Infogérance et conseil en logiciel libre

# Histoire de GIT

Le noyau Linux est un projet libre de grande envergure.

Pour la plus grande partie de sa vie (1991-2002), les modifications étaient transmises sous forme de patchs et d'archives de fichiers.

En 2002, le projet du noyau Linux commença à utiliser un **DVCS** propriétaire appelé BitKeeper.

**DVCS** = Decentralized Version Control System

# Histoire de GIT

En 2005, les relations entre la communauté développant le noyau Linux et la société en charge du développement de BitKeeper furent rompues, et le statut de gratuité de l'outil fut révoqué.

Cela poussa la communauté du développement de Linux (et plus particulièrement Linus Torvalds, le créateur de Linux) à développer son propre outil en se basant sur les leçons apprises lors de l'utilisation de BitKeeper

# Histoire de GIT

Certains des objectifs du nouveau système étaient :

- Vitesse
- conception simple
- support pour les développements non linéaires (milliers de branches parallèles)
- complètement distribué
- capacité à gérer efficacement des projets d'envergure tels que le noyau Linux (vitesse et compacité des données)

## Naissance de GIT



# Vocabulaire

# Commit

Un commit représente la sauvegarde d'un état pour un ensemble de modifications effectuées sur un ou plusieurs fichiers.

Ces modifications peuvent être :

- Une création de fichier / dossier
- Une suppression de fichier / dossier
- Un ajout dans un fichier
- Une modification dans un fichier

**Bonne pratique :**

On rajoutera au commit un « commentaire » pour expliquer ce qui à été effectué.



# Dépôt / Repository

Un dépôt ou repository est le lieu physique où se trouvent les données du projet.

## **Attention :**

L'organisation des fichiers dans un repository n'est pas un système de fichier standard. Même si sur certains gestionnaires on peut les voir comme tel.

Les fichiers doivent être accédés grâce au gestionnaire de versions.

## **Bonus :**

Suivant le gestionnaire de version, le dépôt peut être local ou centralisé.

# Conflit

Si on travaille à plusieurs en même temps sur un même fichier, il est assez courant d'avoir des conflits lors de la fusion des codes des différentes personnes.

Un conflit doit être résolu pour que la fusion soit validée.

# Branche

Une branche est une copie à un moment donné du code.

Il est possible de travailler sur le code de cette nouvelle branche sans toucher le code copié.

Suivant le gestionnaire la création de branche est plus ou moins simple.

**Astuce :**

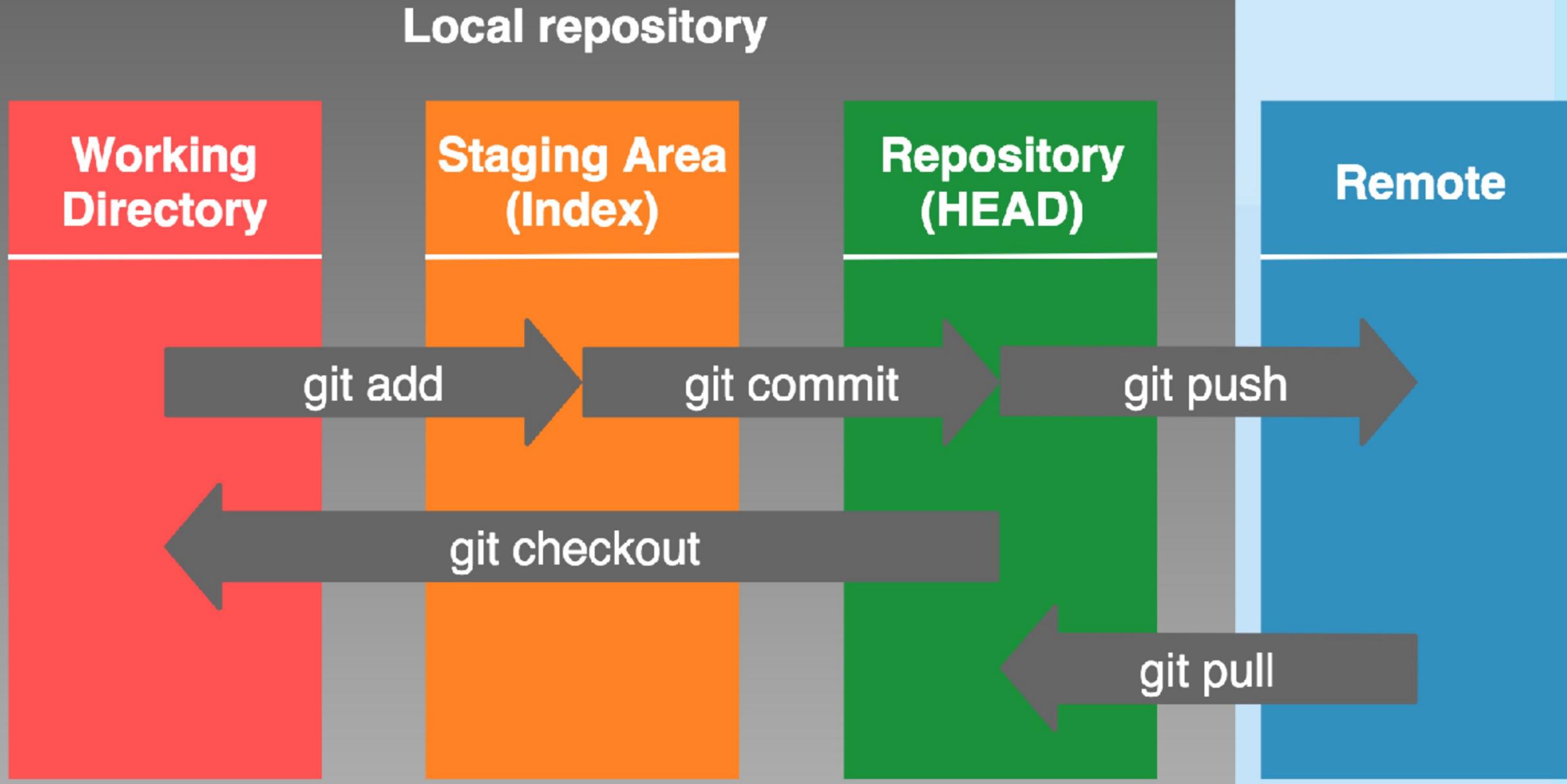
Deux branches peuvent être fusionnées ensemble (merge).

# Tags

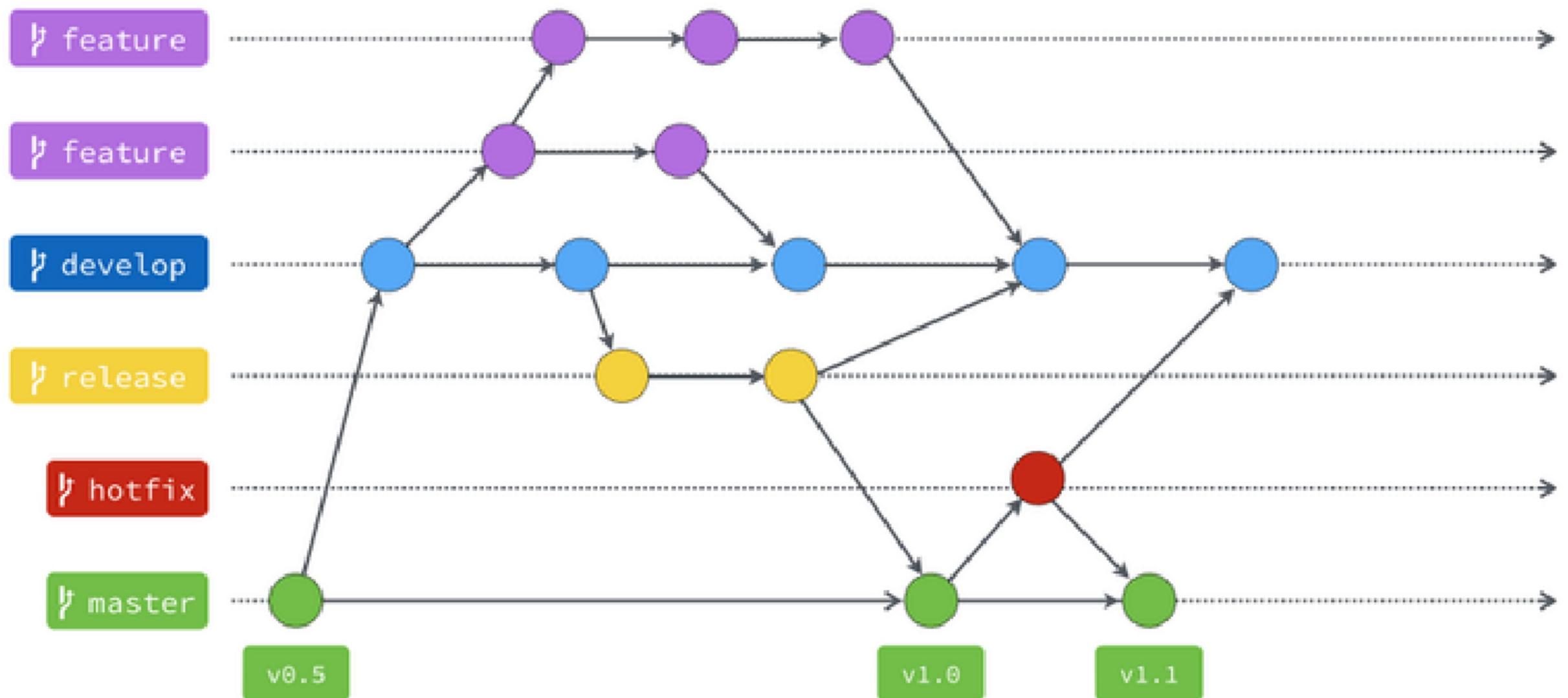
Un tag est une « étiquette » pour identifier l'état du code à un moment donné.

On peut par exemple créer des tags pour les versions 0.1, 0.2 1.0 etc ...

# Workflow



# Workflow



# Installer GIT

**Site officiel :** <https://git-scm.com/>

**Téléchargement :** <https://git-scm.com/downloads>

**Sous Linux, utiliser les dépôts officiels de la distribution :** <https://git-scm.com/download/linux>

# Configurer GIT

## Git global setup

```
git config --global user.name "Jules AGOSTINI"  
git config --global user.email "jules@aukfood.fr"
```

Configuration globale qui se trouve dans le fichier  
**.gitconfig**

Pourra être surchargé dans chaque projet dans le  
fichier **.git/config**

# Configurer GIT

```
1 # utiliser des couleurs dans la sortie de git
2 git config --global color.ui auto
3
4
5 # afficher le journal sur une seule ligne pour chaque validation
6 git config --global format.pretty oneline
```

# Premier projet

# Premier projet

## Push an existing folder

```
cd existing_folder  
git init  
git remote add origin git@gitlab.com:Jagostini/workshop.git  
git add .  
git commit -m "Initial commit"  
git push -u origin master
```

# Premier projet

## Create a new repository

```
git clone git@gitlab.com:Jagostini/workshop.git  
cd workshop  
touch README.md  
git add README.md  
git commit -m "add README"  
git push -u origin master
```

# Premier projet

## Push an existing Git repository

```
cd existing_repo  
git remote rename origin old-origin  
git remote add origin git@gitlab.com:Jagostini/workshop.git  
git push -u origin --all  
git push -u origin --tags
```

# Premier projet

Dans le projet, il y a un répertoire **.git** lequel va contenir toutes les informations permettant la gestion en local du dépôt et la configuration particulière de ce projet.

La commande **git status** permet à tout moment de connaître l'état dans lequel se trouve le projet.

Pour ajouter des fichiers au suivi git on utilise **git add**

# .gitignore

Il peut arriver assez souvent d'avoir besoin d'avoir des fichiers dans son répertoire de travail, mais ne pas vouloir les indexer dans le suivi git.

Par exemple des fichiers de logs, target, ide...

Pour cela git intègre une méthode simple grâce à un fichier **.gitignore**

# Premier commit

# Le problème

```
1 e60930a9 (HEAD -> dev, origin/dev) Modification front
2 49ff1d07 correction d'erreur
3 59a4d13e ajout de log
4 44e2f204 maj conf
5 03d00e2e correction typos
6 10d42648 plein de trucs...
7 bab7be76 linting général
8 4a267eb6 correction encore sur l'écran des users
9 21518e0c correction sur la sélection utilisateur
10 5793cb7a ajout de l'écran de sélection utilisateur
11 a2fdd7d4 PRJ-871
12 85c0f183 traduction
13 c5ff8f0a update configuration de prod
14 c6037d87 correction sur la CI
15 a62f7880 no comment
16 ffefb1b1 désactivation des tests
17 26f99ee0 ...
18 31409e82 ajout de test
19 815485dd Issue #782
20 f9e37f50 monté en version du framework + gestion de dépendances + correction typo
21 1f65e53a Test de la nouvelle solution de sélection
22 ffaf96f7 commit du lundi 5 Mars 2018 - 12h34:25
```

# Conventional-Changelog

## Format

```
1 <type>(<scope>): <subject>
2
3 <body>
4
5 <footer>
```

## Exemple

```
1 feat(users): add user management page
2
3 In order to add users into the system, a
   managment page has been added.
4 The route and services are integrated into
   the UserModule.
5 A mock has been added due to back-end
   limitation.
6
7 Closes #456
```

# Type

- **feat**: Ajout d'une fonctionnalité
- **fix**: Correction
- **perf**: Amélioration des performances
- **docs**: Ajout de documentation
- **style**: Linting du code
- **refactor**: Modification du code, mutualisation et autres modifs afin d'améliorer le projet
- **test**: Ajout de tests unitaires, intégration, end-to-end
- **chore**: Tâche *ménagère*, comme mettre à jour les dépendances, reconfigurer la CI...

# Scope

Pour exemple, sur une application orientée “e-commerce”, on pourrait avoir les scopes suivants :

- homepage
- search
- my-account
- product-details
- order
- admin

# Subject

Quelques conseils pour formater le message :

- Utiliser l'impératif
- Utiliser le présent
- Pas de 1er lettre en majuscule
- Pas de point "." à la fin du sujet
- 50 caractères maximum

# Body

Cela doit contenir le “Pourquoi” du commit.

- Toujours sauter une ligne entre le sujet et le corps
- 72 caractères au maximum.

# Footer

Références vers toutes les issues, User stories ...

# Conventional-Changelog

## Ancienne version

```
1 e60930a9 (HEAD -> dev, origin/dev) Modification front
2 49ff1d07 correction d'erreur
3 59a4d13e ajout de log
4 44e2f204 maj conf
5 03d00e2e correction typos
```

## Nouvelle version

```
1 b1365d1fa (HEAD -> master, origin/master, origin/HEAD) refactor(ivy): remove
directiveRefresh instruction (#22745)
2 4ac606b41 docs(compiler): fix spelling errors (#22704)
3 51027d73c fix(ivy): Update rollup rule to prevent inlining symbols in debug. (#22747)
4 48636f3e8 chore(aio): compute stability and deprecate `@stable` tag (#22674)
5 bd9d4df73 refactor(ivy): remove inputsPropertyName (#22716)
```

# Astuce

```
1 # permet d'ajouter un fichier modifié spécifiquement
2 git add <nomfichier>
3
4 # permet d'ajouter toutes les modifications
5 git add .
6
7 # permet de commenter le commit
8 git commit -m "chore(ci): add jobs"
9
10 # permet en une seule commande de rajouter les modifications et de commenter le
11 git commit -am "chore(ci): add jobs"
```

# Historique

```
1 # permet d'avoir l'historique des commit du projet.  
2 git log
```

L'historique s'affiche dans l'ordre chronologique inversé, du dernier commit au plus ancien.

Chaque commit est identifié avec un identifiant unique, sa somme de contrôle SHA-1, chaîne de 40 caractères hexadécimaux, calculé en fonction du contenu du fichier ou de la structure du répertoire considéré

# Historique

Dans l'historique on voit :

- L'auteur du commit
- La date de celui-ci
- Le commentaire

La commande git log dispose d'une multitude d'options permettant de connaître toutes les infos sur l'évolution du projet.

```
1 # pour afficher les commit des deux dernières semaines :  
2 git log --since 2.weeks
```

# Diff

Si le commit n'a pas été effectué il est possible de connaître la différence entre l'état du dernier commit et l'état actuel des fichiers avec la commande git diff

```
1 # Afficher la différence entre le dernier commit et l'état actuel  
2 git diff
```

# Revenir sur commit

HEAD : désigne le dernier commit ;

HEAD^ : avant-dernier commit ;

HEAD^^ : avant-avant-dernier commit ;

HEAD~2 : avant-avant-dernier commit (notation équivalente) ;

d6d98923868578a7f38dea79833b56d0326fcba1 : indique un numéro de commit précis ;

```
1 # Revient sur le commit précédent
2 git reset HEAD^
3
4 # Supprime en plus de revenir en arrière
5 git reset --hard HEAD^
```

**ATTENTION :** le dernier commit à été annulé ET les modifications faites dans les fichiers sont supprimées

# Taguer un commit

```
1 # Pour tagger le commit courant
2 git tag -a v0.1
3
4 # Pour voir les infos sur le tag
5 git show v0.1
```

Pour tagguer un anciens commit, il faut d'abord récupérer son identifiant (le SHA-1 avec git log), puis le tagguer

```
1 # Tagger un ancien commit
2 git tag -a v0.1 d6d98923868578a7f38dea79833b56d0326fcba1
```

# Ressources

[https://rogerdudler.github.io/git-guide/files/git\\_cheat\\_sheet.pdf](https://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf)

<https://www.cnil.fr/fr/gerer-le-code-source>

<https://github.com/conventional-changelog>

# Merci !

Twitter : @aukfood

Facebook : @aukfood

Site Internet : <https://www.aukfood.fr>

Linkedin : company/aukfood

Mail : contact@aukfood.fr