

# Formation Git

## X - Travailler en équipe

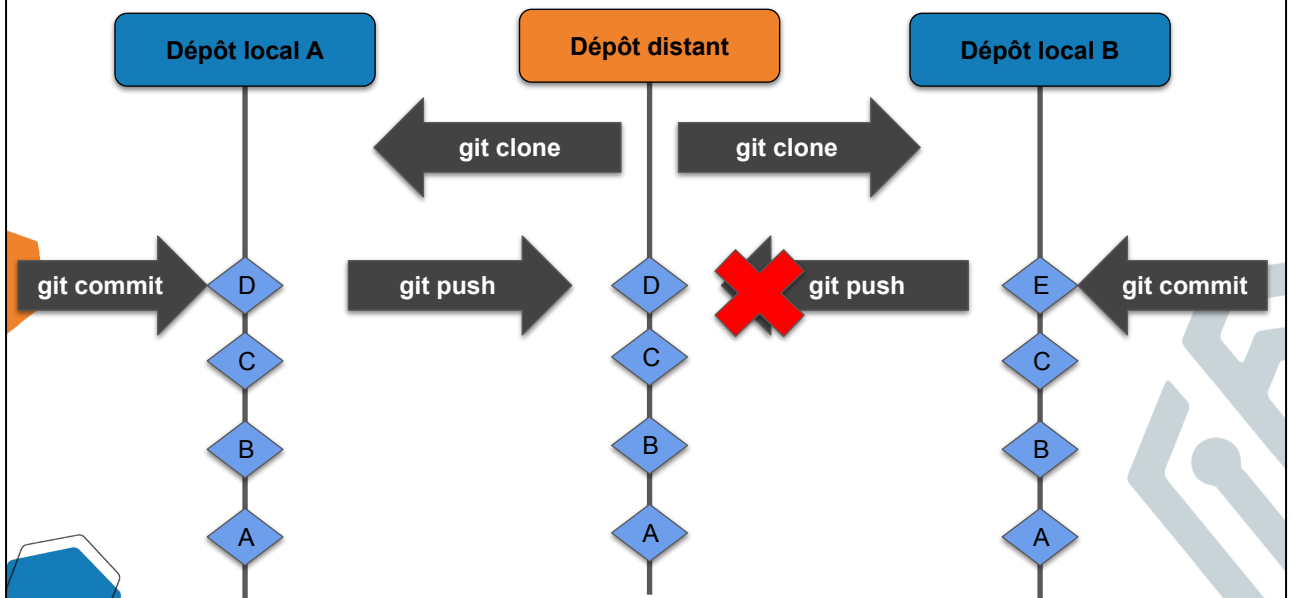


**Arnaud MERCIER**  
arnaud.mercier@hexotech.fr



Nous allons maintenant voir comment travailler en équipe avec Git via un serveur commun

# Travailler avec conflits sur la même branche



Si 2 personnes travaillent sur la même branche (ex: master) et qu'ils modifient cette branche en locale chacun de leur côté, au moment du push le premier n'aura pas de problèmes car il sera en FastForward avec le serveur. En revanche la 2eme personnes aura une erreur lors du push car elle ne sera plus en FastForward avec le serveur. Il y aura une incohérence d'historique (2 versions parallèle de la branche)

## EXEMPLE (faire des modification en même temps sur les 2 repos local

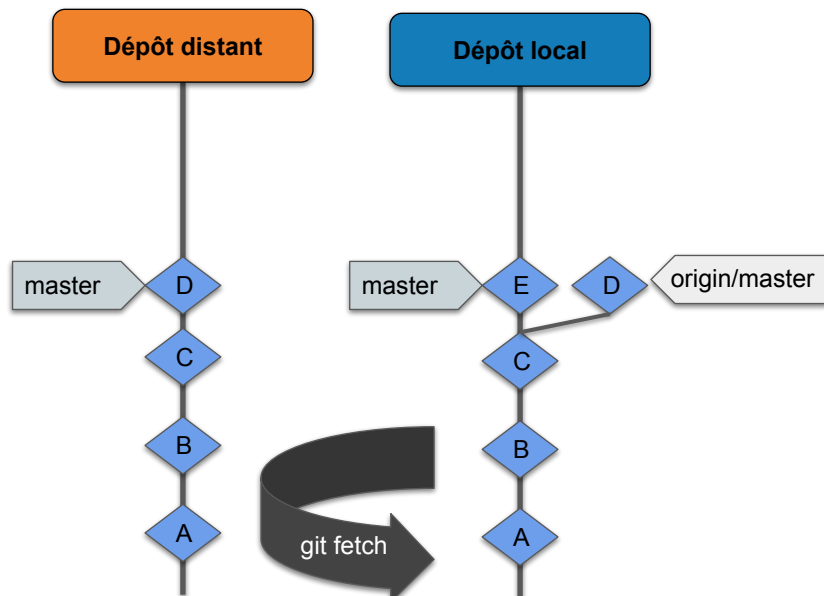
(clone\_serveur):

- `git switch master`
- `git commit -am "modif 1"`
- `git push`

(formation\_git):

- `git switch master`
- `git commit -am "modif 2"`
- `git push` (erreur)

# Git fetch avec conflit



Si on fait un fetch sur le dépôt de la personne qui ne peut pas push, on comprend mieux la situation. Il y a en fait 2 branches:

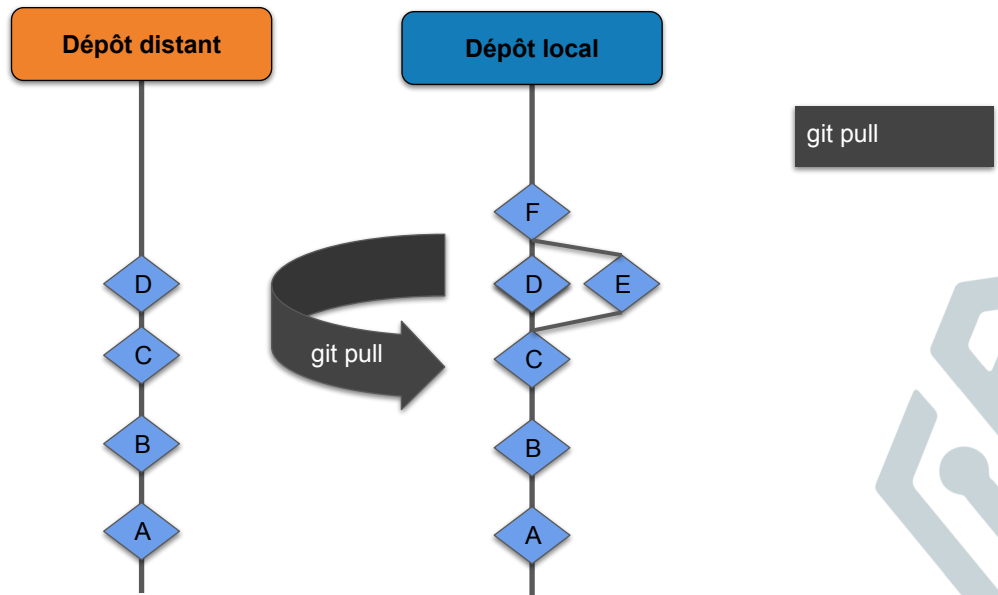
- la branche locale de travail
- la référence distante du remote

Ces deux branches ne sont pas en FastForward. Il est alors possible de faire soit un merge soit un rebase de la branche locale pour se positionner en FastForward

**EXEMPLE (formation\_git):**

- **git fetch**
- **git log --oneline --graph master origin/master**

# Git pull avec conflit



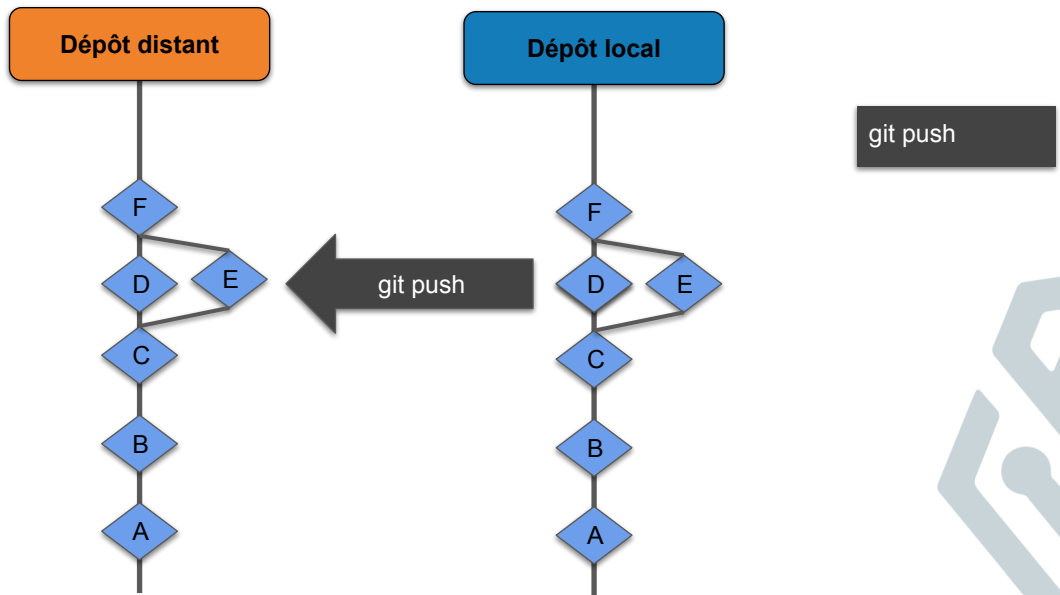
La première solution est le merge. Pour cela soit on le fait à la main, soit on utilise le `git pull`

Attention: Cette méthode fonctionne bien mais pollue grandement votre historique

**EXEMPLE (formation\_git):**

- `git pull`
- `git log --oneline --graph master origin/master`

# Git push après pull



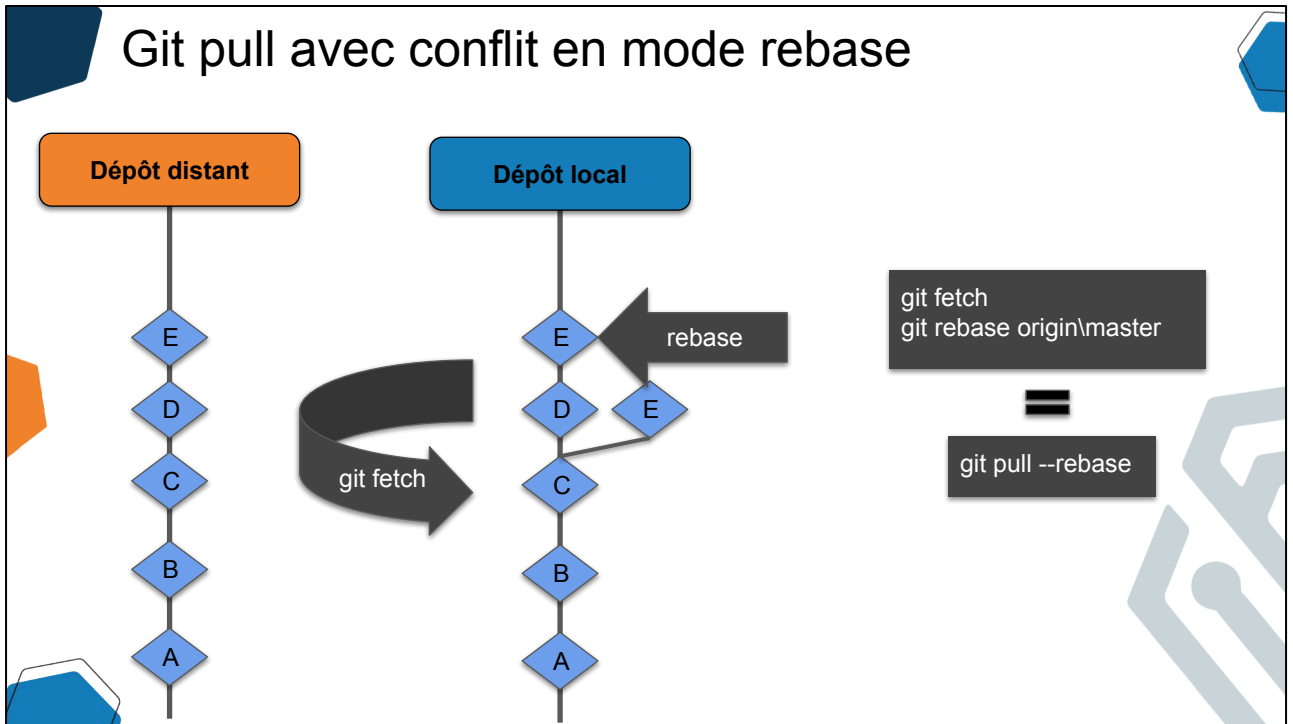
le push fonctionne alors car nous avons un FastForward (non linéaire)

Attention: Cette méthode fonctionne bien mais pollue grandement votre historique

**EXEMPLE (formation\_git):**

- **git push**
- **git log --oneline --graph master origin/master**

# Git pull avec conflit en mode rebase



Il est possible d'éviter le commit de merge en indiquant l'option `--rebase` à la commande `git pull`.

Cette méthode permet d'avoir un historique plus simple à relire. Elle est fortement recommandée dans le cas où plusieurs personnes modifient la même branche de travail.

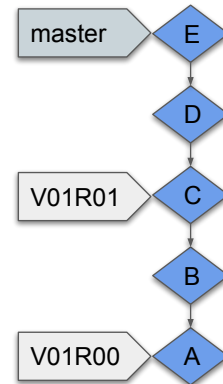
## EXEMPLE (formation\_git):

- `git reset --hard ORIG_HEAD`
- `git pull --rebase`
- `git log --oneline --graph master origin/master`

# Centralized Workflow

## Branches permanentes

- **Master:** contient l'ensemble des commits et les releases (tags)



**Recommandé pour les projets simples avec uniquement si un seul développeur**

Principe: Tout le monde travail sur la branche master et push sur la branche master. il faut donc gérer les conflits, potentiellement, à chaque push.

Avantages: c'est le workflow le plus simple à mettre en place.

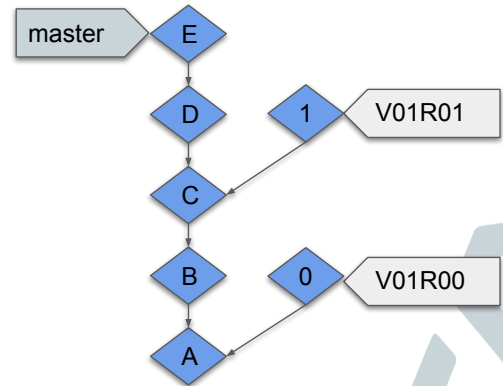
Limitations: uniquement adapté à un développeur sur le projet.

# Release Workflow

## Branches permanentes

- **Master:** contient l'ensemble des commits

Les releases (tags) sont sur des commits spécifique partants de master



**Projets simples avec procédure de release (montée de version ou freeze des dépendances)**

Principe: Tout le monde travail sur la branche master et push sur la branche master. il faut donc gérer les conflits, potentiellement, à chaque push. Quand une release est à générer (actions spécifique comme montée de version ou freeze des dépendances), on part du commit souhaité et on réalise un commit orphelin sur lequel on pose le tag

Avantages: workflow simple avec gestion d'une release basique

Limitations: Il est adapté aux très petites équipes qui travaillent dans des parties du code bien distinctes.



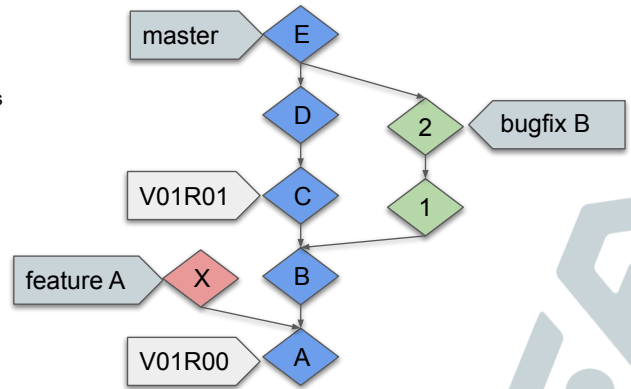
# Feature Branch Workflow

## Branches permanentes

- **Master**: contient les release de l'application et les merges

## Branches à durées limitées

- **Feature/xxx** : branche de fonctionnalités
- **BugFix/xxx**: branches de correction de bugs

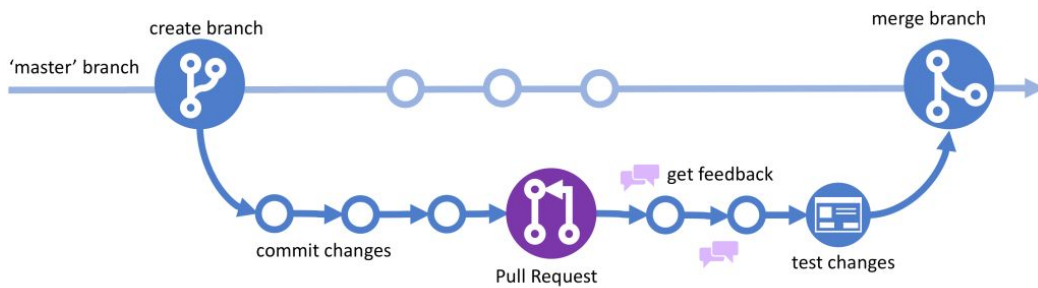


**Principe:** Chaque développeur travaille dans une branche de développement en local (feature ou bugfix). Une fois sèche et stable, il la verse dans le master via un rebase de préférence et push ses modification sur la master distante. Enfin une Pull request est réalisé sur github pour merger la branche de travail

**Avantages:** c'est une amélioration du workflow centralisé qui limite grandement les cas de conflits.

**Limitations:** Il est adapté au projets avec déploiements fréquents (ex: saas) mais ne permet pas de gérer des release en parallèle des intégrations

# Feature Branch Workflow (GitHub flow)



Le workflow est recommandé par Github et est fortement intégré sur la plateforme

# Git Flow

Vincent Driessen



<https://nvie.com/>

## Branches à durées limitées

- **Feature:** une branche par fonctionnalités
- **Release:** branche de stabilisation avant release
- **BugFix:** branches de correction de bugs (develop)
- **HotFix:** branches de correction de bugs (release)

## Branches permanentes

- **Master:** contient les release de l'application
- **develop:** contient les développements
- **Support:** branches de support d'une ancienne version

Projets complexe avec cycles de développement et de release (voir de valide)

GitFlow à été créé en janvier 2010 par Vincent Driessen qui travail notamment beaucoup sur du développement web

Même si il existe des outils qui permetant d'appliquer GitFlow de manière assisté, notamment sous forme de plugin git, GitFlow est plus comme un framework qui définit un modèle de création de branche strict conçu autour de la livraison de projet. Il n'ajoute finalement aucun nouveau concept et aucune nouvelles commande au git de base. Les commandes Gitflow sont finalement des macros de commandes git classiques.

Dans son principe, gitflow attributs des rôles très spécifique a chaque branche et détermine les interactions qu'elle doivent avoir entre elle. cad quelle branches doivent être merge entre elles

Gitflow, comporte deux grands types de branches:

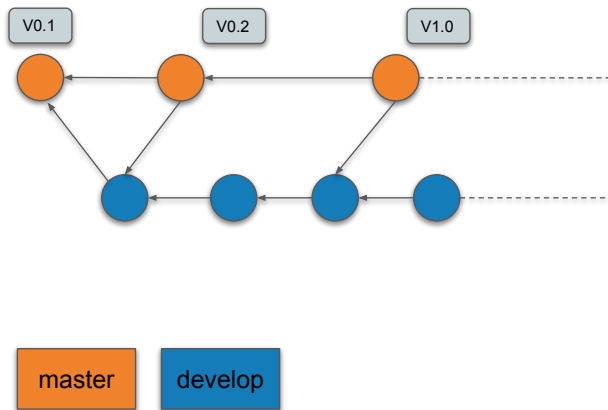
Branches a durées limités

- Feature branches (une branche par fonctionnalités)
- Release branches (branche de stabilisation avant release)
- BugFix branches (branche de correction de bug)
- HotFix branches (branche de correction de bugs)

### Branches permanentes

- Master branche (contient les release de l'application)
- develop branche (contient les développements)
- Support branches (branche de support d'un ancienne version)

# Branches develop et master



```
git checkout -b develop
```

```
outil git
```

```
git flow init
```

```
plugin gitflow
```

```
git branch
* develop
master
```

La base de GitFlow, est constitué de deux branches principales(master et develop) contrairement au workflows plus classiques qui utilisent uniquement une branche master comme unique base.

- La branche principale (master) stocke l'historique officiel des versions, comme on peut le voir ici avec les tags
- La branche de développement (develop) sert de branche d'intégration pour les fonctionnalités.

Pour mettre en place cette base, il faut:

- 1- Avoir un dépôt Git classique (avec une branche master)
- 2- Ajouter la branche de développement (develop)

Une solution très simple consiste à créer une branche de développement (develop) vide en local et d'en faire un push vers le serveur :

```
$ git checkout -b develop
```

Il est également possible d'utiliser l'outil gitflow

```
$ git flow init
```

cette commande vas alors vous créer automatiquement les branches et vous

positionner sur develop

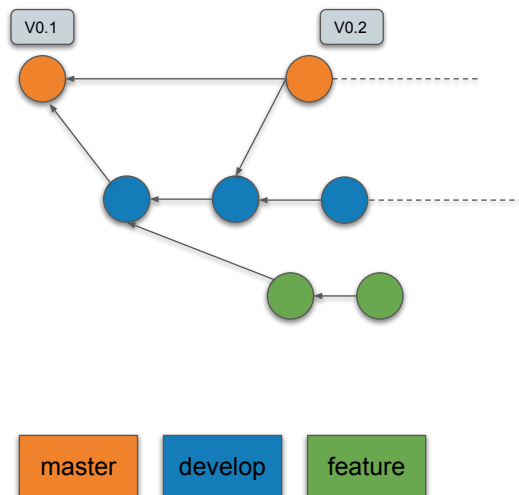
**Attention**, lors du git flow init, celui-ci vous demande les préfixe pour les différentes branches temporaire. Je vous conseil de laisser les valeurs par défaut

Au finale, on se retrouve avec deux branches

develop

master

# Branches de fonctionnalité [START]



```
git checkout develop  
git checkout -b feature/[name]
```

outil git

=

```
git flow feature start [name]
```

plugin gitflow

```
git branch  
*feature/[name]  
develop  
master
```

Comme pour les autres workflows git, on crée une branche par nouvelle fonctionnalités que l'on souhaite ajouter. Mais à la différences de la plupart des workflows, ici, les branche de fonctionnalités (features) ne partent pas de master mais de develop.

Cette branche peut être poussé vers le dépôt centralisé en vue d'une sauvegarde/collaboration.

## Création d'une branche de fonctionnalité

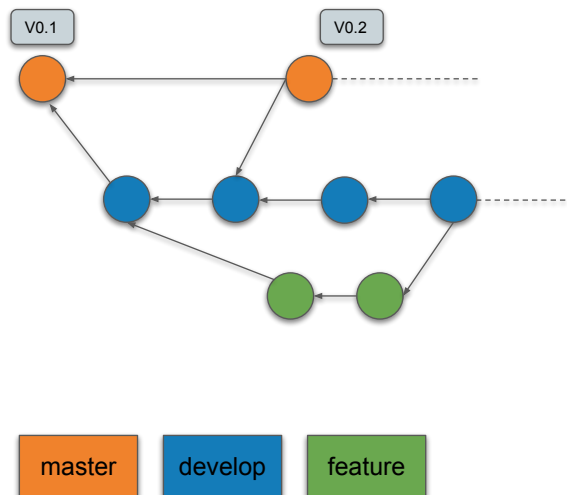
```
$ git checkout develop
```

```
$ git checkout -b feature/name
```

ou

Attention, gitflow ajoute automatiquement le prefix release/, donc inutile de l'indiquer  
`git flow feature start [name]`

# Branches de fonctionnalité [FINISH]



```
git checkout develop
git merge --no-ff feature/[name]
git branch -d feature/[name]
git push -d origin feature/[name]
```

outil git

=

```
git flow feature finish [name]
```

plugin gitflow

Une fois la feature terminée, il convient de merger la branche de fonctionnalité (feature) dans la branche de développement (develop).

Les fonctionnalités ne communiquent jamais directement avec la branche principale (master).

Sans les extensions git-flow :

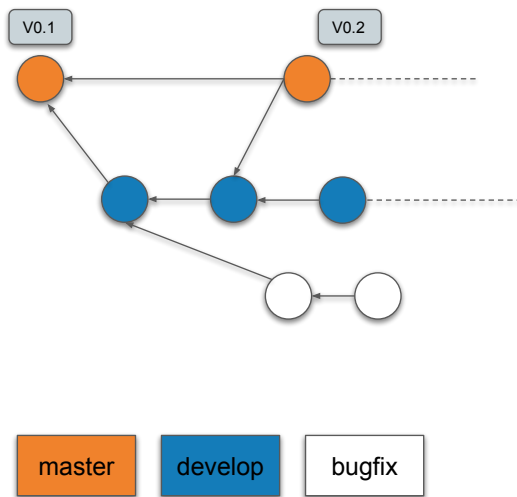
```
$ git checkout develop
$ git merge --no-ff feature/[name]
$ git branch -d feature/[name]
$ git push -d origin feature/[name]
```

Avec les extensions git-flow :

```
$ git flow feature finish [name]
```



# Branches de BugFix [START]



```
git checkout develop  
git checkout -b bugfix/[name]
```

outil git

=

```
git flow bugfix start [name]
```

plugin gitflow

```
git branch  
*bugfix/[name]  
develop  
master
```

les branches bugfix sont identique aux branches de release, mais leur objectif est de faire des correction de bug et non des évolutions

sans gitflow

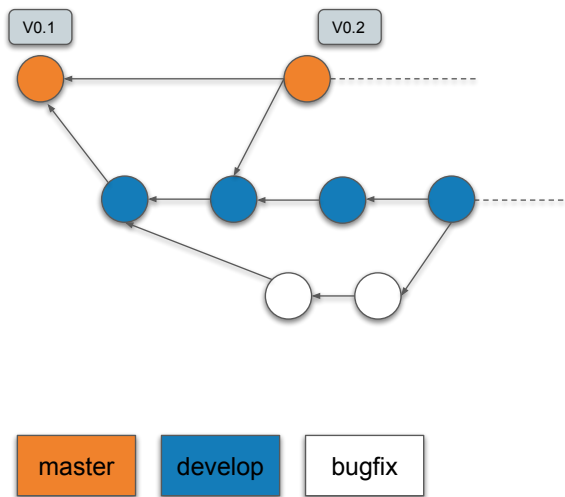
```
$ git checkout develop
```

```
$ git checkout -b bugfix/[name]
```

avec gitflow

```
$ git flow bugfix start [name]
```

# Branches de BugFix [FINISH]



```
git checkout develop
git merge --no-ff bugfix/[name]
git branch -d bugfix/[name]
git push -d origin bugfix/[name]
```

outil git

=

```
git flow bugfix finish [name]
```

plugin gitflow

Une fois la feature terminée, il convient de merger la branche de fonctionnalité (feature) dans la branche de développement (develop).

Les fonctionnalités ne communiquent jamais directement avec la branche principale (master).

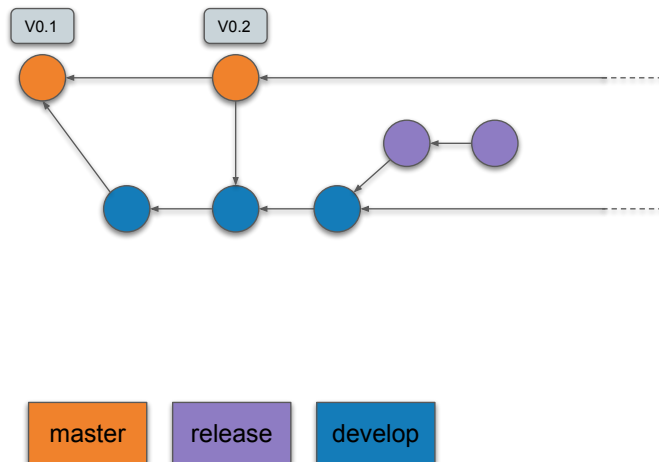
Sans les extensions git-flow :

```
$ git checkout develop
$ git merge --no-ff feature/[name]
$ git branch -d feature/[name]
$ git push -d origin feature/[name]
```

Avec les extensions git-flow :

```
$ git flow feature finish [name]
```

# Branches de livraison [START]



```
git checkout develop
git checkout -b release/[version]
```

outil git

=

```
git flow release start [version]
```

plugin gitflow

Lorsque les fonctionnalités pour la prochaine version sont toutes mergées dans develop, il faut préparer la livraison. Pour cela, on ne merge pas directement la branche de développement (develop) dans la branche master.

On préfère passer par une branche de stabilisation (release) à partir de la branche de développement (develop).

La création de cette branche marque le début du cycle de livraison suivant. Ainsi, aucune fonctionnalité supplémentaire ne pourra être ajoutée. Dorénavant, cette branche sera uniquement dédiée aux corrections de bugs, à la génération de documentation et à d'autres tâches axées sur la livraison.

Le numéro de version à livrer est déterminé à ce moment-là et il est utilisé comme nom de branche préfixé par 'release/'

En utilisant une branche dédiée pour préparer les livraisons, une équipe peut affiner la version actuelle tandis qu'une autre équipe continue de travailler sur les fonctionnalités de la version suivante.

Cela permet de bien découper les phases de développement et notamment de sécher une version avant de la livrer. Ici on a une vraie étape de stabilisation, ce qui évite de tomber dans le piège où on se dit "bon j'ai le temps d'ajouter une autre petite

fonctionnalité”.

Sans les extensions git-flow :

```
$ git checkout develop
```

```
$ git checkout -b release/[version]
```

Lorsque vous utilisez des extensions git-flow :

```
$ git flow release start [version]
```

**Important:** c'est à ce moment là que l'on viens faire évoluer le numéro de version dans les sources: par exemple via un script comme c'est le cas avec npm version

# Branches de livraison [FINISH]

```
graph LR; V0.1((V0.1)) --> D1(( )); D1 --> V0.2((V0.2)); V0.2 --> D2(( )); D2 --> V1.0((V1.0)); V1.0 --> D3(( )); D3 --> D4(( )); D4 --> D5(( )); D5 --> D6(( )); D6 --> D7(( )); D7 --> D8(( )); D8 --> D9(( )); D9 --> D10(( )); D10 --> D11(( )); D11 --> D12(( )); D12 --> D13(( )); D13 --> D14(( )); D14 --> D15(( )); D15 --> D16(( )); D16 --> D17(( )); D17 --> D18(( )); D18 --> D19(( )); D19 --> D20(( )); D20 --> D21(( )); D21 --> D22(( )); D22 --> D23(( )); D23 --> D24(( )); D24 --> D25(( )); D25 --> D26(( )); D26 --> D27(( )); D27 --> D28(( )); D28 --> D29(( )); D29 --> D30(( )); D30 --> D31(( )); D31 --> D32(( )); D32 --> D33(( )); D33 --> D34(( )); D34 --> D35(( )); D35 --> D36(( )); D36 --> D37(( )); D37 --> D38(( )); D38 --> D39(( )); D39 --> D40(( )); D40 --> D41(( )); D41 --> D42(( )); D42 --> D43(( )); D43 --> D44(( )); D44 --> D45(( )); D45 --> D46(( )); D46 --> D47(( )); D47 --> D48(( )); D48 --> D49(( )); D49 --> D50(( )); D50 --> D51(( )); D51 --> D52(( )); D52 --> D53(( )); D53 --> D54(( )); D54 --> D55(( )); D55 --> D56(( )); D56 --> D57(( )); D57 --> D58(( )); D58 --> D59(( )); D59 --> D60(( )); D60 --> D61(( )); D61 --> D62(( )); D62 --> D63(( )); D63 --> D64(( )); D64 --> D65(( )); D65 --> D66(( )); D66 --> D67(( )); D67 --> D68(( )); D68 --> D69(( )); D69 --> D70(( )); D70 --> D71(( )); D71 --> D72(( )); D72 --> D73(( )); D73 --> D74(( )); D74 --> D75(( )); D75 --> D76(( )); D76 --> D77(( )); D77 --> D78(( )); D78 --> D79(( )); D79 --> D80(( )); D80 --> D81(( )); D81 --> D82(( )); D82 --> D83(( )); D83 --> D84(( )); D84 --> D85(( )); D85 --> D86(( )); D86 --> D87(( )); D87 --> D88(( )); D88 --> D89(( )); D89 --> D90(( )); D90 --> D91(( )); D91 --> D92(( )); D92 --> D93(( )); D93 --> D94(( )); D94 --> D95(( )); D95 --> D96(( )); D96 --> D97(( )); D97 --> D98(( )); D98 --> D99(( )); D99 --> D100(( )); D100 --> D101(( )); D101 --> D102(( )); D102 --> D103(( )); D103 --> D104(( )); D104 --> D105(( )); D105 --> D106(( )); D106 --> D107(( )); D107 --> D108(( )); D108 --> D109(( )); D109 --> D110(( )); D110 --> D111(( )); D111 --> D112(( )); D112 --> D113(( )); D113 --> D114(( )); D114 --> D115(( )); D115 --> D116(( )); D116 --> D117(( )); D117 --> D118(( )); D118 --> D119(( )); D119 --> D120(( )); D120 --> D121(( )); D121 --> D122(( )); D122 --> D123(( )); D123 --> D124(( )); D124 --> D125(( )); D125 --> D126(( )); D126 --> D127(( )); D127 --> D128(( )); D128 --> D129(( )); D129 --> D130(( )); D130 --> D131(( )); D131 --> D132(( )); D132 --> D133(( )); D133 --> D134(( )); D134 --> D135(( )); D135 --> D136(( )); D136 --> D137(( )); D137 --> D138(( )); D138 --> D139(( )); D139 --> D140(( )); D140 --> D141(( )); D141 --> D142(( )); D142 --> D143(( )); D143 --> D144(( )); D144 --> D145(( )); D145 --> D146(( )); D146 --> D147(( )); D147 --> D148(( )); D148 --> D149(( )); D149 --> D150(( )); D150 --> D151(( )); D151 --> D152(( )); D152 --> D153(( )); D153 --> D154(( )); D154 --> D155(( )); D155 --> D156(( )); D156 --> D157(( )); D157 --> D158(( )); D158 --> D159(( )); D159 --> D160(( )); D160 --> D161(( )); D161 --> D162(( )); D162 --> D163(( )); D163 --> D164(( )); D164 --> D165(( )); D165 --> D166(( )); D166 --> D167(( )); D167 --> D168(( )); D168 --> D169(( )); D169 --> D170(( )); D170 --> D171(( )); D171 --> D172(( )); D172 --> D173(( )); D173 --> D174(( )); D174 --> D175(( )); D175 --> D176(( )); D176 --> D177(( )); D177 --> D178(( )); D178 --> D179(( )); D179 --> D180(( )); D180 --> D181(( )); D181 --> D182(( )); D182 --> D183(( )); D183 --> D184(( )); D184 --> D185(( )); D185 --> D186(( )); D186 --> D187(( )); D187 --> D188(( )); D188 --> D189(( )); D189 --> D190(( )); D190 --> D191(( )); D191 --> D192(( )); D192 --> D193(( )); D193 --> D194(( )); D194 --> D195(( )); D195 --> D196(( )); D196 --> D197(( )); D197 --> D198(( )); D198 --> D199(( )); D199 --> D200(( )); D200 --> D201(( )); D201 --> D202(( )); D202 --> D203(( )); D203 --> D204(( )); D204 --> D205(( )); D205 --> D206(( )); D206 --> D207(( )); D207 --> D208(( )); D208 --> D209(( )); D209 --> D210(( )); D210 --> D211(( )); D211 --> D212(( )); D212 --> D213(( )); D213 --> D214(( )); D214 --> D215(( )); D215 --> D216(( )); D216 --> D217(( )); D217 --> D218(( )); D218 --> D219(( )); D219 --> D220(( )); D220 --> D221(( )); D221 --> D222(( )); D222 --> D223(( )); D223 --> D224(( )); D224 --> D225(( )); D225 --> D226(( )); D226 --> D227(( )); D227 --> D228(( )); D228 --> D229(( )); D229 --> D230(( )); D230 --> D231(( )); D231 --> D232(( )); D232 --> D233(( )); D233 --> D234(( )); D234 --> D235(( )); D235 --> D236(( )); D236 --> D237(( )); D237 --> D238(( )); D238 --> D239(( )); D239 --> D240(( )); D240 --> D241(( )); D241 --> D242(( )); D242 --> D243(( )); D243 --> D244(( )); D244 --> D245(( )); D245 --> D246(( )); D246 --> D247(( )); D247 --> D248(( )); D248 --> D249(( )); D249 --> D250(( )); D250 --> D251(( )); D251 --> D252(( )); D252 --> D253(( )); D253 --> D254(( )); D254 --> D255(( )); D255 --> D256(( )); D256 --> D257(( )); D257 --> D258(( )); D258 --> D259(( )); D259 --> D260(( )); D260 --> D261(( )); D261 --> D262(( )); D262 --> D263(( )); D263 --> D264(( )); D264 --> D265(( )); D265 --> D266(( )); D266 --> D267(( )); D267 --> D268(( )); D268 --> D269(( )); D269 --> D270(( )); D270 --> D271(( )); D271 --> D272(( )); D272 --> D273(( )); D273 --> D274(( )); D274 --> D275(( )); D275 --> D276(( )); D276 --> D277(( )); D277 --> D278(( )); D278 --> D279(( )); D279 --> D280(( )); D280 --> D281(( )); D281 --> D282(( )); D282 --> D283(( )); D283 --> D284(( )); D284 --> D285(( )); D285 --> D286(( )); D286 --> D287(( )); D287 --> D288(( )); D288 --> D289(( )); D289 --> D290(( )); D290 --> D291(( )); D291 --> D292(( )); D292 --> D293(( )); D293 --> D294(( )); D294 --> D295(( )); D295 --> D296(( )); D296 --> D297(( )); D297 --> D
```

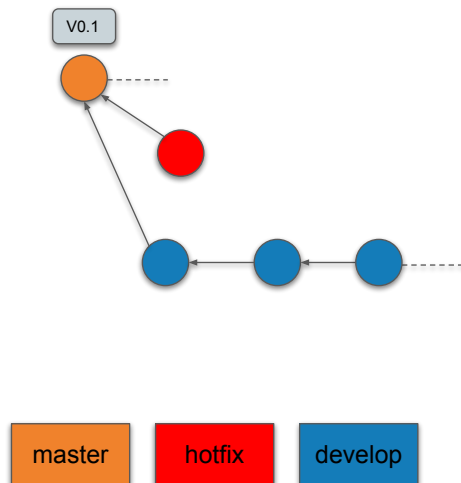
Une fois les merges réalisés, la branche de version (release) est supprimée.

```
$ git checkout master
$ git merge release/[version]
$ git tag [version] -am"message"
```

```
$ git checkout develop
$ git merge release/[version]
$ git branch -d release/[version]
$ git push -d origin release/[version]
```

```
$ git flow release finish [version]
```

# Branches hotfix [START]



```
git checkout master  
git checkout -b hotfix/[name]
```

outils git

```
git flow hotfix start [name]
```

outils gitflow

Les branches de maintenance ou « hotfix » sont utilisées pour appliquer rapidement des patches aux versions de production.

Les branches de maintenance (hotfix) sont très similaires aux branches de version (release) et de fonctionnalité (feature), si ce n'est qu'elles sont basées sur la branche principale (master) et non sur la branche de développement (develop).

Il est recommandé de consacrer une ligne de développement aux corrections de bugs : votre équipe pourra ainsi résoudre les problèmes sans devoir interrompre le reste du workflow ou attendre le cycle de livraison suivant.

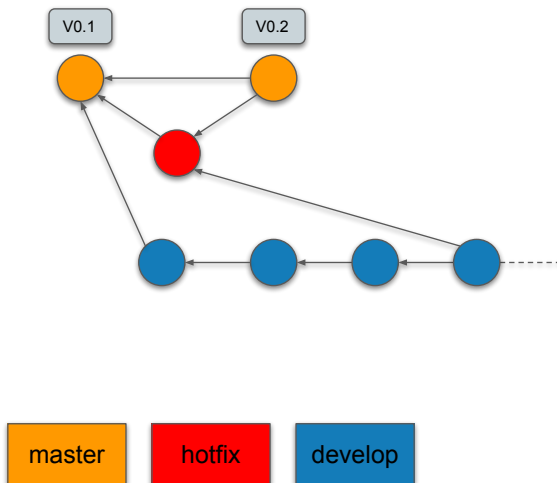
Sans les extensions git-flow :

```
$ git checkout master  
$ git checkout -b hotfix/[name]
```

Lorsque vous utilisez des extensions git-flow :

```
$ git flow hotfix start [name]
```

# Branches hotfix [FINISH]



```
git checkout master
git merge hotfix/[name]
git tag [tag] -am "message"
```

```
git checkout develop
git merge hotfix/[name]
git branch -d hotfix/[name]
git push -d origin hotfix/[name]
```

outils git

=

```
git flow hotfix finish hotfix/[name]
```

outils gitflow

Une fois les corrections terminées, il faut faire le merge dans la branche master

Attention à bien penser à faire le merge également dans la branche de développement (develop) pour faire bénéficier le reste de l'équipe de ces corrections

Une fois les merges réalisés, on peut supprimer la branche et poser le tag sur le commit de merge dans master

Sans le plugin gitFlow

```
$ git checkout master
```

```
$ git merge hotfix/[name]
```

```
$ git tag [tag] -am "message"
```

```
$ git checkout develop
```

```
$ git merge hotfix/[name]
```

```
$ git branch -d hotfix/[name]
```

```
$ git push -d origin hotfix/[name]
```

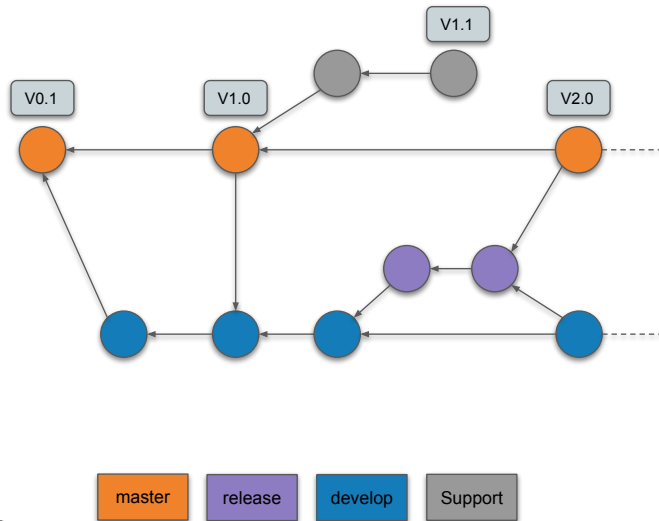
Avec le plugin GitFlow

```
$ git flow hotfix finish hotfix_branch
```





# Branches Support



```
git checkout [version]  
git checkout -b support/[version]
```

outils git

```
git flow support start [version]
```

outils gitflow

```
git checkout develop  
git cherry-pick [sha1]
```

Lorsque la vie du projet nécessite de générer une nouvelle release patché d'une version plus ancienne du produit, il n'est alors pas possible de passer par la branche hotfix. En effet, cela signifie générer lors du merge de cette branche, un nouveau commit dans la branche master. Or ce commit devrait, en théorie se trouver entre la version que l'on patch et les autres version plus récentes du produit. Cela correspond alors à une réécriture de l'histoire, chose qu'il faut clairement éviter.

La solution est donc de créer une branche de support qui porte le nom de la version que l'on maintient, préfixé par support/

Sans GitFlow

```
$ git checkout [version]  
$ git checkout -b support/[version]
```

Avec gitflow

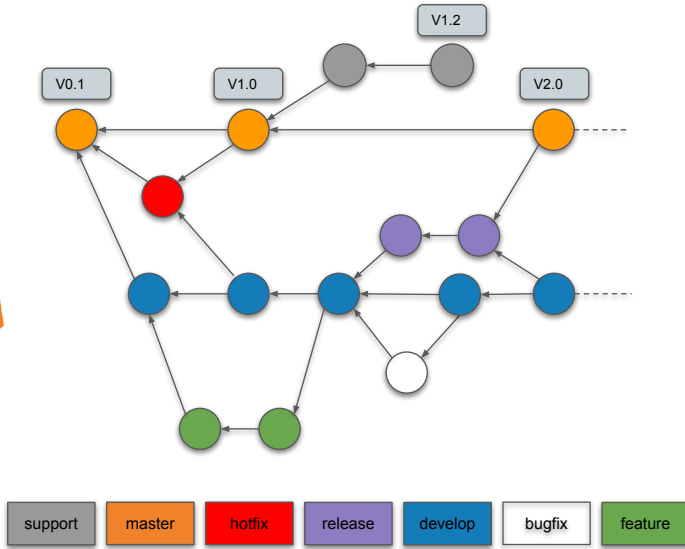
```
$ git flow support start [version]
```

Il est alors possible de reporter les évolutions/correction de cette branche dans la branche develop via

```
$ git checkout develop  
$ git cherry-pick [sha1]
```



# Gitflow vue d'ensemble



- 1- Une branche develop est créée à partir de master.
- 2- Les branches feature et bugfix sont créées à partir de develop.
- 3- Quand une branche feature est terminée, elle est mergée dans la branche develop, puis supprimée
- 4- Une branche release est créée à partir de develop.
- 5- Lorsque la branche release est terminée, elle est mergée dans les branches develop et master (création tag), puis supprimée
- 6- Une branche hotfix est créée à partir de master.
- 7- Une fois la branche hotfix terminée, elle est mergée dans les branches develop et master (création tag), puis supprimée
- 8- Une branche support est créée à partir d'un tag sur master

Pour résumer:

GitFlow est un workflow pour Git qui donne des rôles spécifiques à chaque branche, cela permet de découper l'historique en différentes phases bien distinctes. De plus Gitflow limite les interactions entre les branches, voici les règles à respecter:

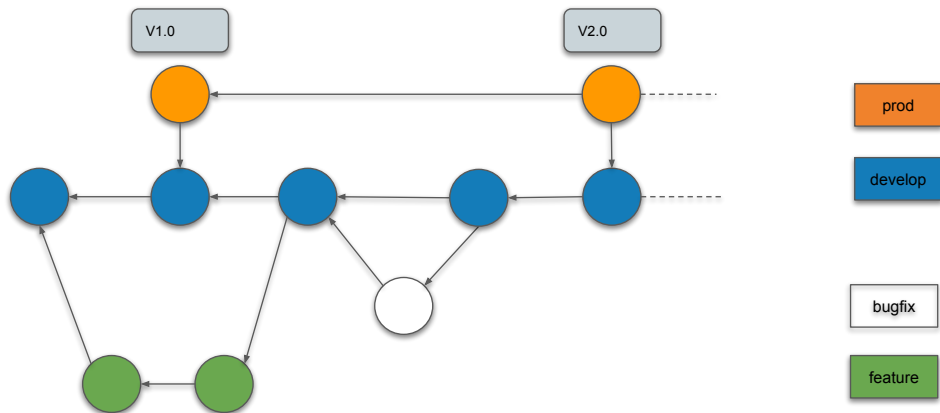
- 1- Une branche de développement (develop) est créée à partir de master.
- 2- Les branches de fonctionnalité (feature) sont créées à partir de develop.
- 3- Quand une branche de fonctionnalité (feature) est terminée, elle est mergée dans la branche de développement (develop), puis supprimée
- 4- Une branche de version (release) est créée à partir de develop.
- 5- Lorsque la branche de version (release) est terminée, elle est mergée dans les branches de développement (develop) et principale (master), puis supprimée
- 6- Une branche de maintenance (hotfix) est créée à partir de master.

7- Une fois la branche de maintenance (hotfix) terminée, elle est mergée dans les branches de développement (develop) et principale (master), puis supprimée

8- Une branche support est créée à partir d'un tag sur master

Gitflow n'ajoute pas de commandes ou de notions au git de base, mais vous pouvez utiliser le plugin Git: gitflow pour simplifier son application

# Production workflow



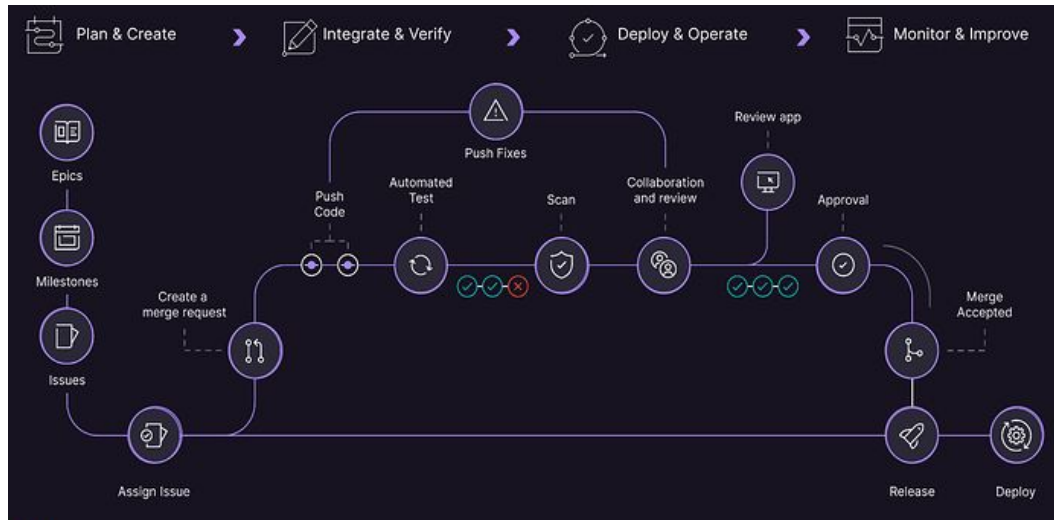
Version simplifié du gitflow qui peut évoluer vers un gitflow si besoin

**Principe:** Chaque développeur travaille dans une branche de développement en local. Une fois sèche et stable, il la verse dans le master via un rebase de préférence et push ses modification sur la master distante

**Avantages:** c'est une amélioration du workflow centralisé qui limite grandement les cas de conflits.

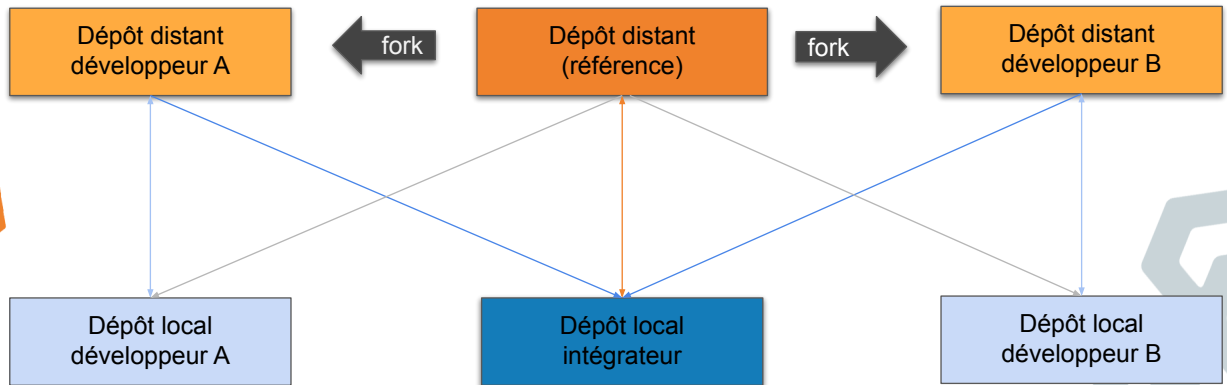
**Limitations:** Il est adapté aux équipes de moyenne taille.

# Production workflow (Gitlab flow)



Ce workflow est recommandé par gitlab et est intégré directement dans leur plateforme. Notamment via leur système de merge request

# Intégration ou Forking Workflow



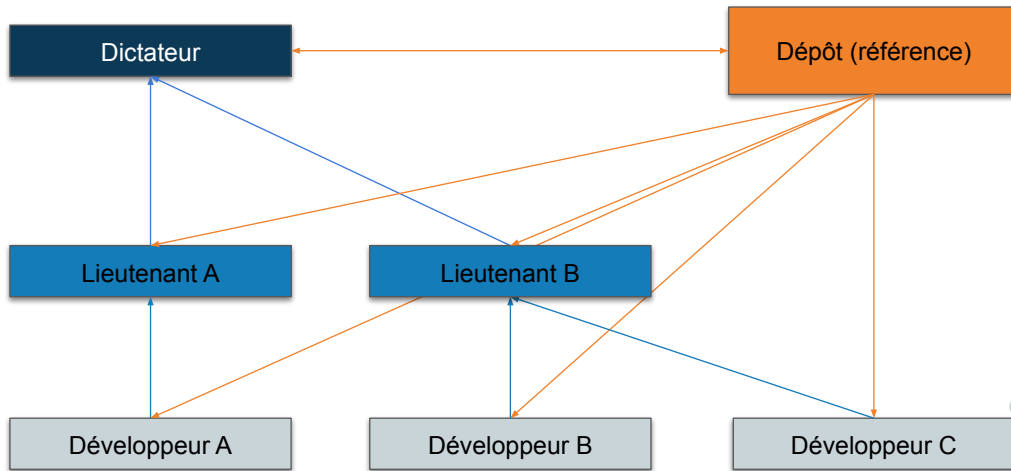
Principe: Ce workflow est utilisé dans les projets opensource. Le dépôt distant n'est accessible en écriture que par le propriétaire et des personnes autorisés. chaque développeurs, non autorisés et souhaitant participer au projet doivent:

- 1- Duplique (fork) le dépôt distant. Cela lui permet d'avoir un dépôt distant avec les accès en écriture.
- 2- Cloner sa copie du dépôt distant en local.
- 3- Ajouter un remote vers le dépôt d'origine (celui en lecture seul). Cela permet de se mettre à jour
- 4- Créer une branche de développement en local pour sa contribution
- 5- Faire ses commits et push sur son dépôt distant
- 6- Envoyer un pull request au propriétaire du dépôt d'origine pour qu'il puisse intégrer les modifications dans son dépôt

Avantages: Pas d'accès directe au dépôt. relecture de code par l'expert. intégration des modification quand on le souhaite

Limitations: Workflow long à mettre en place et qui peut paraître complexe. De plus il demande une bonne maîtrise de Git

# Dictator Workflow



C'est le même principe global que l'intégration workflow mais avec un étage de plus. Cela permet de décharger l'intégrateur sur de très gros projet. Chaque lieutenant sera responsable de l'intégration des contribution dans son périmètre d'expertise.

Le dictateur, lui, se charge de l'intégration finale en prenant en compte les intégrations des lieutenants

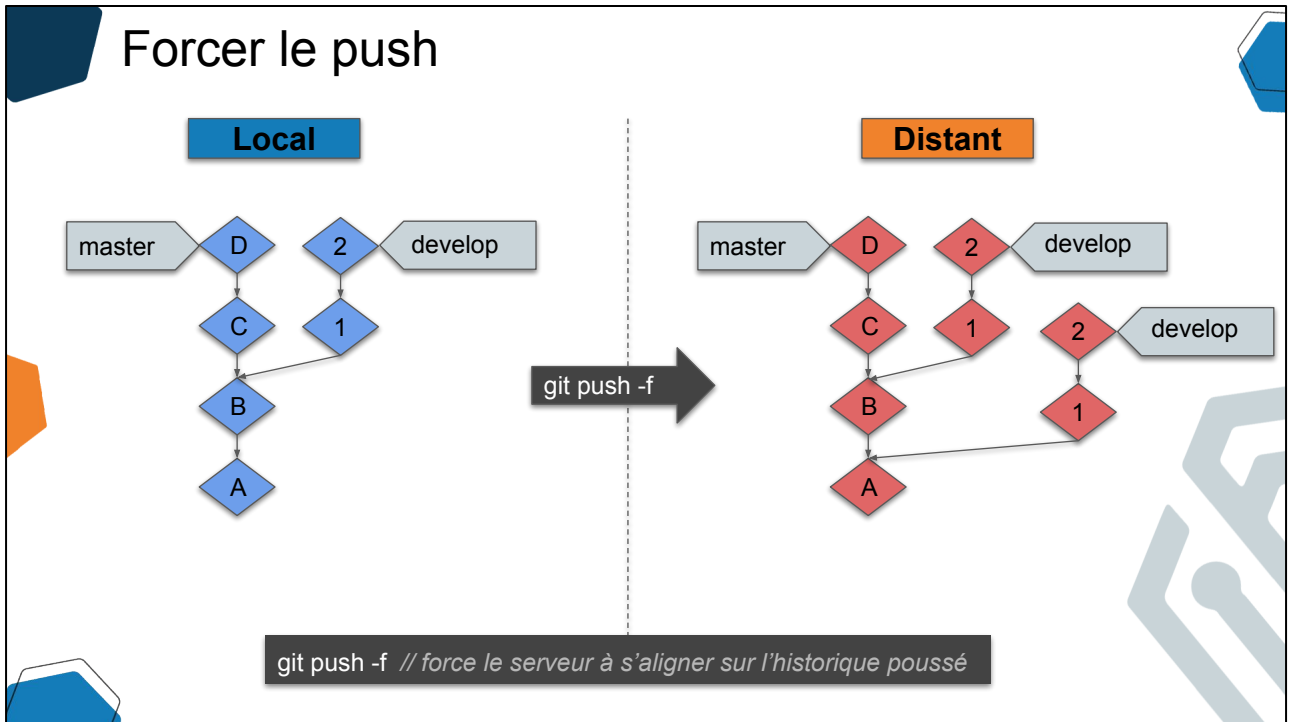


## Votre Workflow



Il est impératif d'adapter votre propre workflow répondre au mieux à votre besoin.

# Forcer le push



Si l'historique est réécrit en local, il n'est pas possible de push simplement cette réécriture. En effet, la branche local ne sera plus en fast forward par rapport à la branche distante.

Git cherchera alors à vous faire fusionner l'historique de la branche local et celui de la branche distante. Ce n'est pas ce que nous voulons. Nous cherchons ici à imposer notre historique réécrit.

Il faut alors push force (il faut avoir le droit de le faire sur le serveur).

INFO: Le push force est plutôt réservé aux branches temporaires

## EXEMPLE

(clone\_serveur):

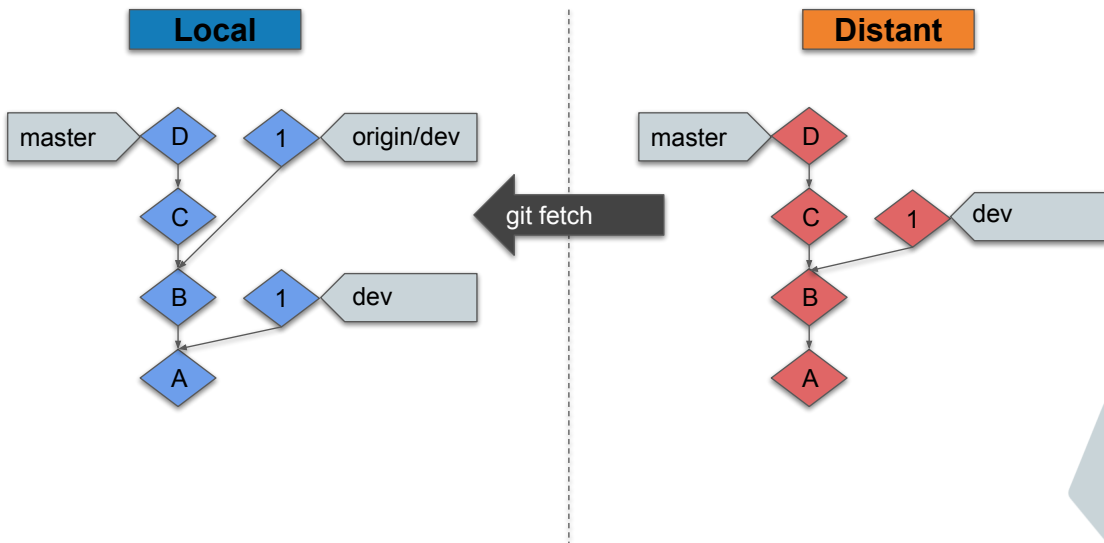
- `git switch -c develop`
- `git commit -am "modif A"`
- `git commit -am "modif B"`
- `git push -u origin develop`

(formation\_git):

- `git fetch`
- `git switch develop`
- `git rebase -i origin/master (squash)`
- `git push (erreur)`

- **git push -f**

## Fetch et historique modifié

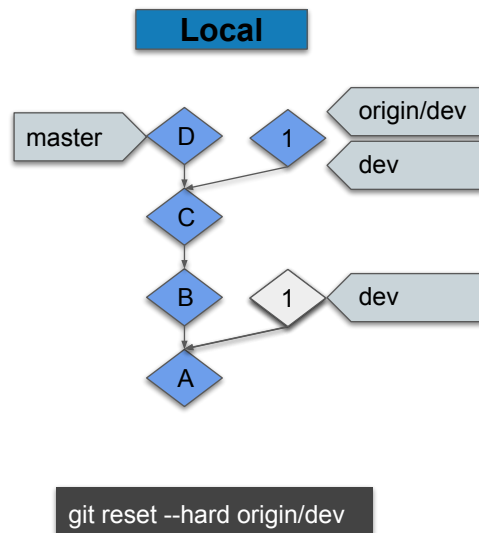


Si une réécriture d'historique est push sur le serveur, pour les autres personnes, il y aura une incohérence des historiques entre la version local et la version serveur

### EXEMPLE (clone\_serveur):

- `git fetch`
- `gitk develop origin/develop`

# Forcer le dépôt local



Il faut alors forcer l'historique local a se conformer a l'historique distante. Dans le cas contraire, git cherchera a faire un merge entre les 2 version de la branche.

Ici nous voulons nous forcer sur l'historique du serveur.

Pour cela, il faut utiliser la commande `git reset` pour forcer la branche local a pointer sur le même commit que la branche distante

## EXEMPLE (clone\_serveur):

- **git pull (merge)**
- **gitk develop**
- **git reset --hard origin/develop**
- **gitk develop**

# TP 10.1

## Scénario

Vous voulez faire un commit de vos modifications, mais pas de chance, votre collègue vient de faire justement un push sur le même fichier que vous. Au moment de faire le git pull, vous avez des conflits de merge

## Objectifs

1- Sur le dépôt local "clone\_formation\_git", réaliser une modification dans hello.html. Ajouter une section "mes commandes Git favorites" et y inscrire une commande.

2- Faire le commit et push

3- Sur le dépôt local "formation\_git", réaliser une modification dans hello.html. Ajouter une section "mes commandes Git favorites" et y inscrire une commande différente du point 2

4- Récupérer les modifications présentes sur le serveur et les merger avec les vôtres

5- Faire le push

6- Sur le dépôt local "clone\_formation\_git", ajouter une nouvelle commande favorite et faire le commit

7- Faire un pull qui conserve un historique linéaire

Solution:

1- `<div><h1> Mes commandes </h1><li> git status </li></div>`

2- git commit et git push

3- `<div><h1> Mes commandes </h1><li> git log </li></div>`

4- git pull

5- git push

6- git commit et git push

7- git pull --rebase

## TP 10.2


### Scénario

Pour éviter au maximum des conflits entre développeurs, vous décidez de partir en branche pour faire vos développements.

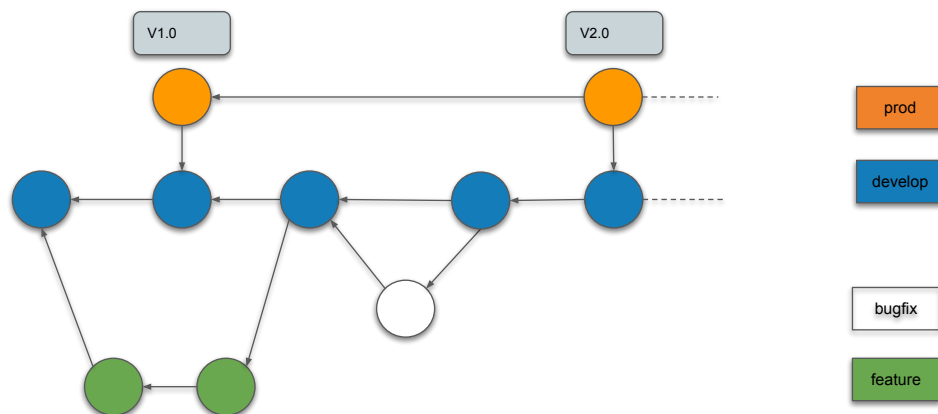
### Objectifs

- 1- Sur le dépôt local "formation\_git", partir en branche depuis le tag LOT1 du nom de MEDAL
- 2- Dans cette branche, ajouter la médaille d'or et faire le commit
- 3- Faire un push de la branche MEDAL
- 4- Vérifier que celle-ci est bien présente sur le serveur
- 5- Dans le dépôt local "clone\_formation\_git", récupérer la branche MEDAL
- 6- Faire le rebase + merge de la branche MEDAL dans master et faire le push
- 7- Poser le tag LOT2 et faire le push

Solution:

- 1- git checkout Site\_V03 et git checkout -b MEDAL
- 2- 
- 3- git push
- 4- gitweb
- 5- git pull + git checkout MEDAL
- 6- git rebase master + git checkout master + git merge MEDAL
- 7- git tag SITE\_V04

# Bilan



Nous avons vu ensemble comment travailler efficacement en équipe autour d'un dépôt Git