

Introduction à Git (et Gitlab)

Travaux Pratiques

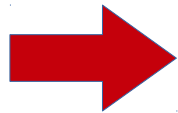
Robin Passama
LIRMM – CNRS/UM

Plan

- **Installation**
- Tutoriel
 - Un premier projet
 - Travail collaboratif
 - Conseils et astuces

Installation

- Configurer git sur son PC
 - Configurer les informations sur l'utilisateur
 - `git config --global user.name "Robin Passama"`
 - `git config --global user.email "passama@lirmm.fr"`



L'email sera utilisé par Gitlab pour identifier les commits

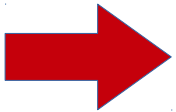
Installation

- Configurer SSH sur son PC

- `cd ~/.ssh`

- `ssh-keygen -b 2048 -t rsa`

Entrer un nom pour la clé (par exemple GitlabKey)



Entrer un mot de passe

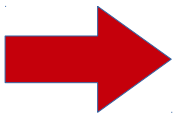
Résultat : **des clés privées et publiques sont générées**

- `chmod 600 ~/.ssh/<your_private_key> #celle sans le .pub`

- Pour ne plus avoir à taper le mot de passe de votre clé SSH

- `ssh-agent`

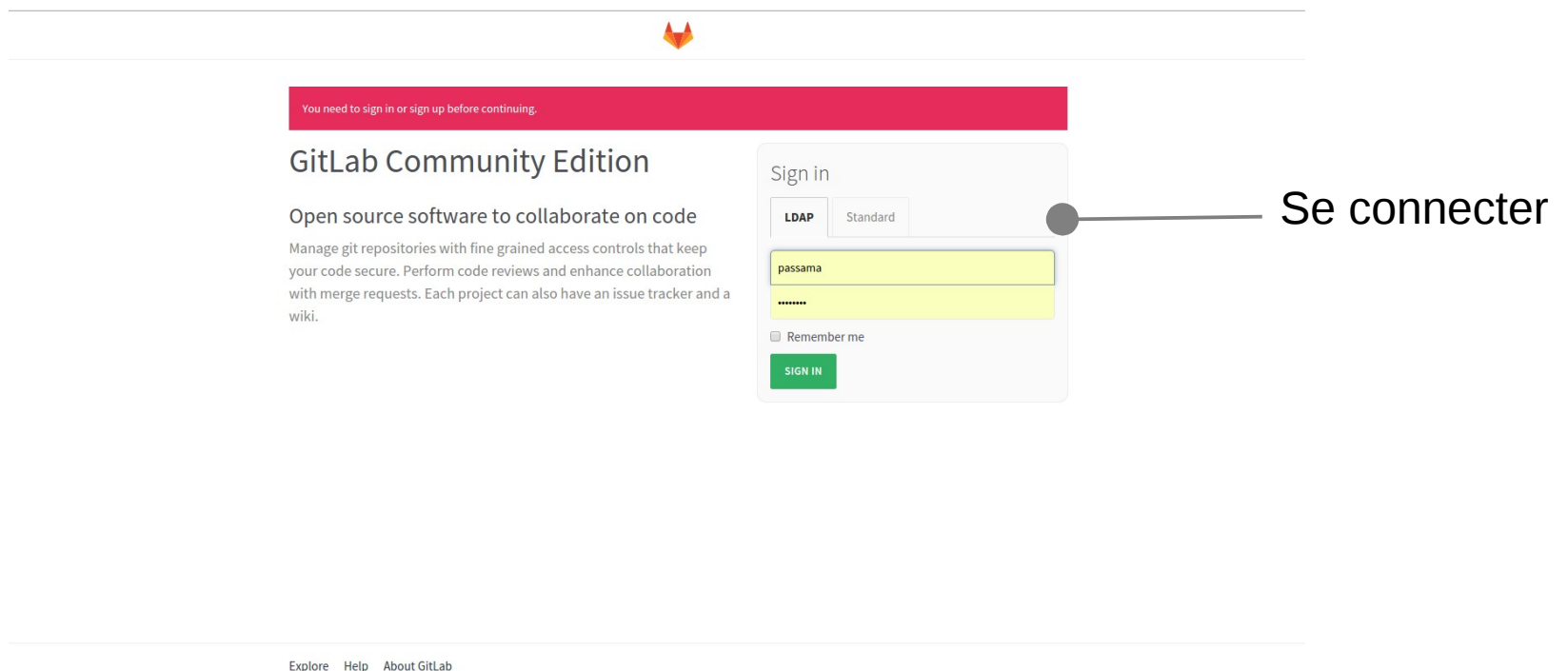
- `ssh-add ~/.ssh/<your_private_key>`



Entrer votre mot de passe

Installation

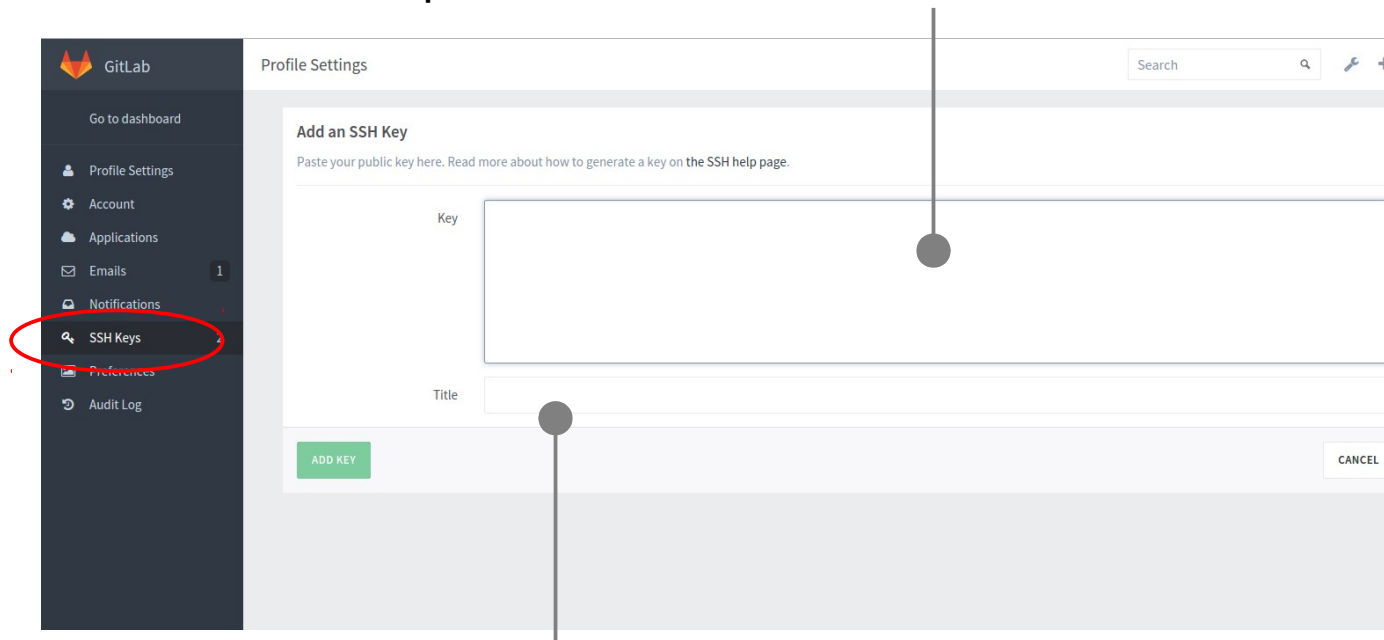
- Se connecter à GitLab
 - Pour les membres de l'IUT: utiliser vos identifiants <https://gitlabinfo.iutmontp.univ-montp2.fr/>



Installation

- Pour ajouter votre clé SSH sur le serveur
 - Aller dans les paramètres de votre compte

Copier/coller le contenu de votre clé SSH ici



The screenshot shows the GitLab web interface. On the left is a dark sidebar with the GitLab logo and a list of navigation items: 'Go to dashboard', 'Profile Settings', 'Account', 'Applications', 'Emails', 'Notifications', 'SSH Keys' (which is circled in red), 'Preferences', and 'Audit Log'. The main content area is titled 'Profile Settings' and contains a section 'Add an SSH Key'. Below this section is a text input field labeled 'Key' and a smaller text input field labeled 'Title'. At the bottom of the form are two buttons: a green 'ADD KEY' button and a white 'CANCEL' button. A line with a dot points from the text 'Copier/coller le contenu de votre clé SSH ici' to the 'Key' input field. Another line with a dot points from the text 'Un titre pour identifier votre machine (e.g. machines de l'IUT)' to the 'Title' input field.

Un titre pour identifier votre machine (e.g. machines de l'IUT)

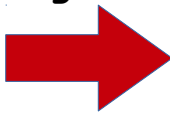
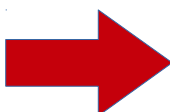
Plan

- Installation
- **Tutoriel**
 - **Un premier projet**
 - Travail collaboratif
 - Conseils et astuces

Premier projet

- Dans Gitlab:
 - Chercher le dépôt ***git-first-example*** de Robin Passama dans la barre de recherche.
 - Ouvrez le projet et récupérez l'adresse du dépôt.
- Sur votre PC, ouvrez un terminal:
 - `cd <somewhere>`
 - `git clone git@gitlabinfo.iutmontp.univ-montp2.fr:passama/git-first-example.git`
 - `cd git-first-example && ls -la`
- Ouvrez README.md et regardez son contenu

Premier projet

- Pour lister les branches:
 - `git branch`
 Seule la branche `master` apparaît (branche locale par défaut)
 - `git branch -a`
 Les branches `origin/master` et `origin/dev` apparaissent aussi (elles sont aussi sur GitLab)
 - La branche courante est indiquée par un astérisque
- Pour automatiser l'affichage de la branche courante voir
 - <https://gitlabinfo.iutmontp.univ-montp2.fr/passama/git-first-example/wikis/home>

Premier projet

- Travaillons sur la branche **dev**
 - `git checkout dev` #change current branch
 - `ls -la` #il y a un fichier de plus
- Ouvrez README.md et observez que le contenu a changé
- Le rôle du fichier `.gitignore` :
 - Exclure du gestionnaire de version les fichiers qui correspondent au motif (ici les fichiers finissant par '~').
 - S'applique aux sous-dossiers...
 - Mais les sous-dossiers peuvent avoir leur `.gitignore`.
- Git gère aussi les versions du fichier `.gitignore`

Premier projet

- Pour visualiser votre dépôt local
 - `git log` #version texte
 - `gitg` #un outil graphique
- Pour visualiser le dépôt serveur sur GitLab
 - Cliquer sur le menu “Repository” dans le panneau latéral gauche, puis:
 - Cliquer sur l’onglet “Graph” (~= `gitg`), ou
 - Cliquer sur l’onglet “Commits” (~= `git log`) après avoir sélectionné la branche.

Premier projet

- Vérifier le status de votre dépôt local

- **git status**

- Sur la branche dev

- Votre branche est à jour avec 'origin/dev'.

- ...

- Rien besoin de faire pour l'instant...

Premier projet

- Modifiez le contenu de README.md en ajoutant le texte que vous souhaitez
- Vérifiez à nouveau le statut de votre dépôt:

- **git status**

Sur la branche dev

Votre branche est à jour avec 'origin/dev'.

Modifications qui ne seront pas validées :

(utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)

(utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

modifié: README.md

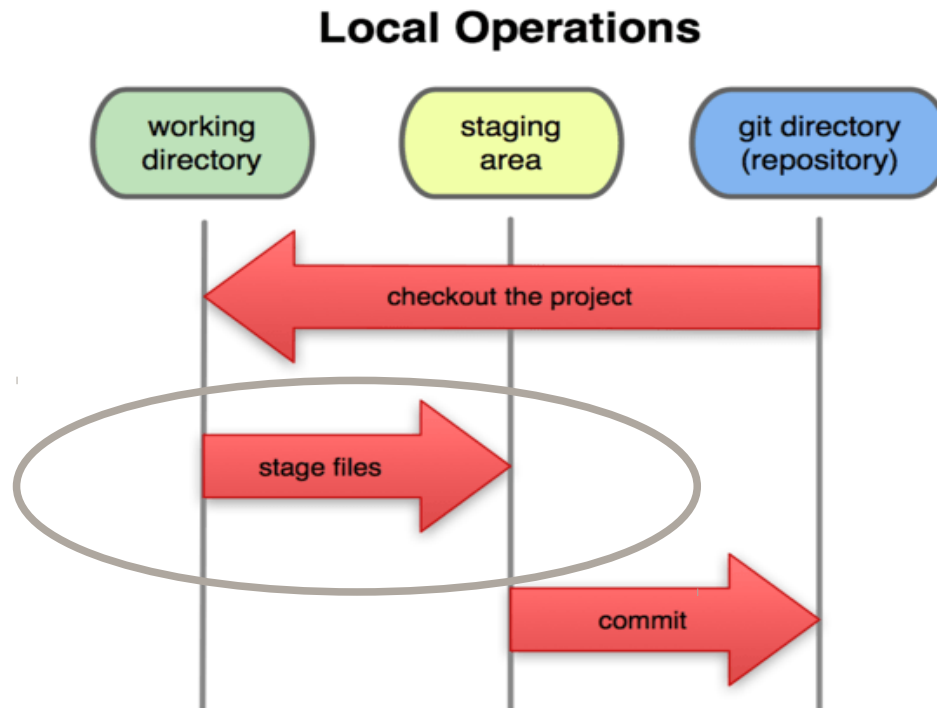
aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")

Premier projet

- Pour voir les modifications depuis votre dernier commit
 - `git diff`
 - '+' indique une ligne ajoutée dans le fichier.
 - '-' indique une ligne enlevée dans le fichier.

Premier projet

- Sélectionnons les modifications à indexer (stage)
 - `git add --all` #indexe toutes les modifications



Premier projet

- Sélectionnons les modifications à indexer (stage)
 - `git add --all` #indexe toutes les modifications
- Vérifions le statut de notre dépôt:

- `git status`

Sur la branche dev

Votre branche est à jour avec 'origin/dev'.

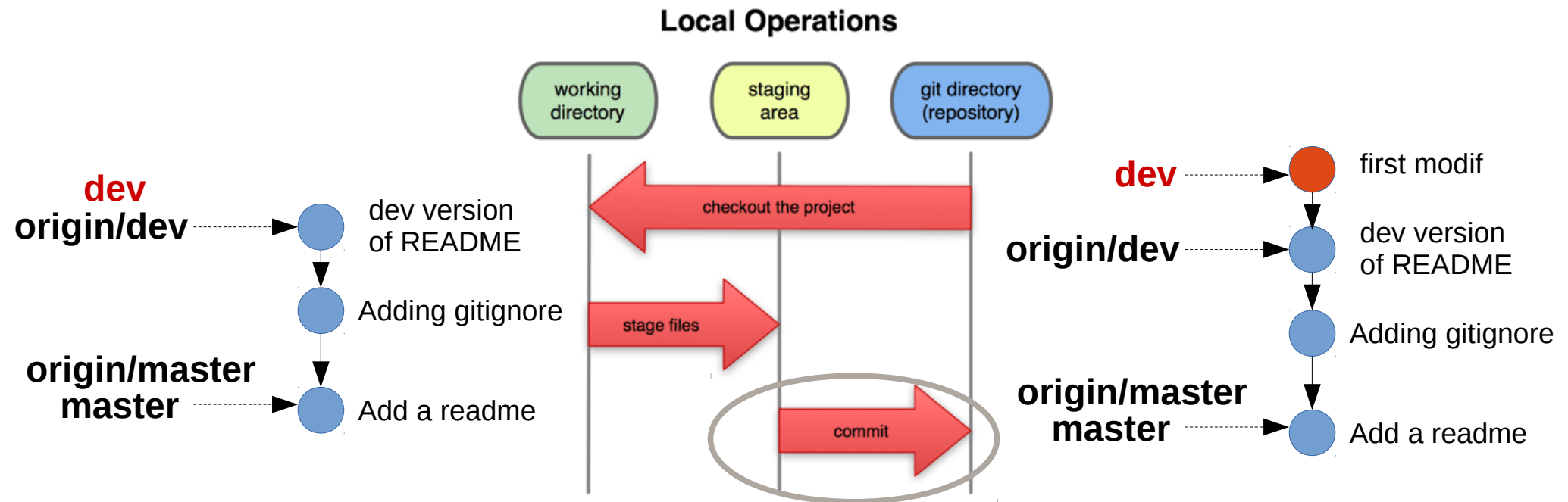
Modifications qui seront validées :

(utilisez "`git reset HEAD <fichier>...`" pour désindexer)

modifié: **README.md**

Premier projet

- Validons maintenant nos modifications (commit)



Premier projet

- Validons nos modifications (commit)
 - `git commit -m "first modif"`
`#les modifications indexées sont validées`
- Vérifions le statut de notre dépôt:
 - `git status`
Sur la branche dev

Votre branche est en avance sur 'origin/dev' de 1 commit.

(utilisez "git push" pour publier vos commits locaux)

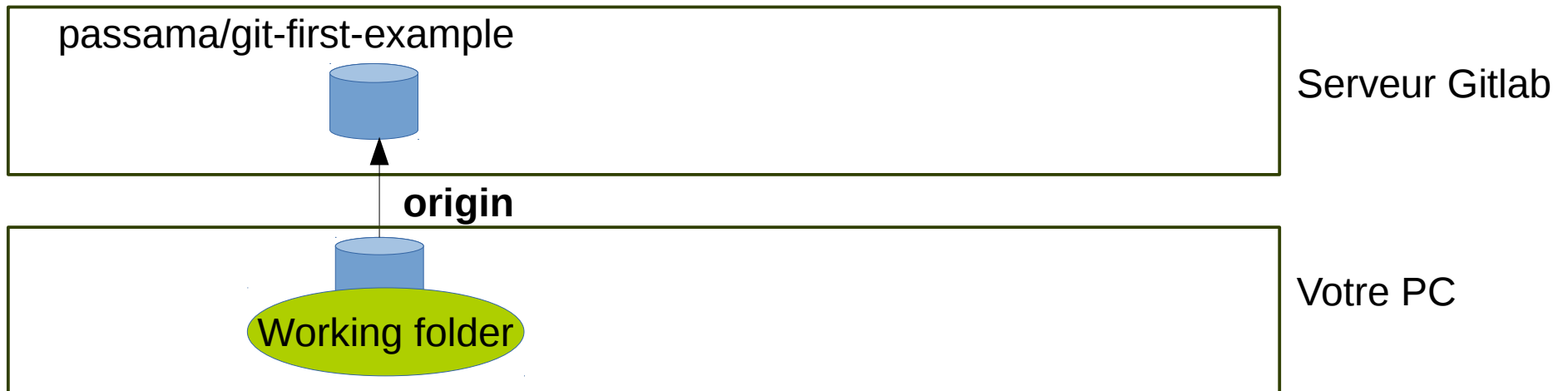
rien à valider, la copie de travail est propre
- Utiliser aussi `gitg` pour voir le nouveau statut du dépôt

Premier projet

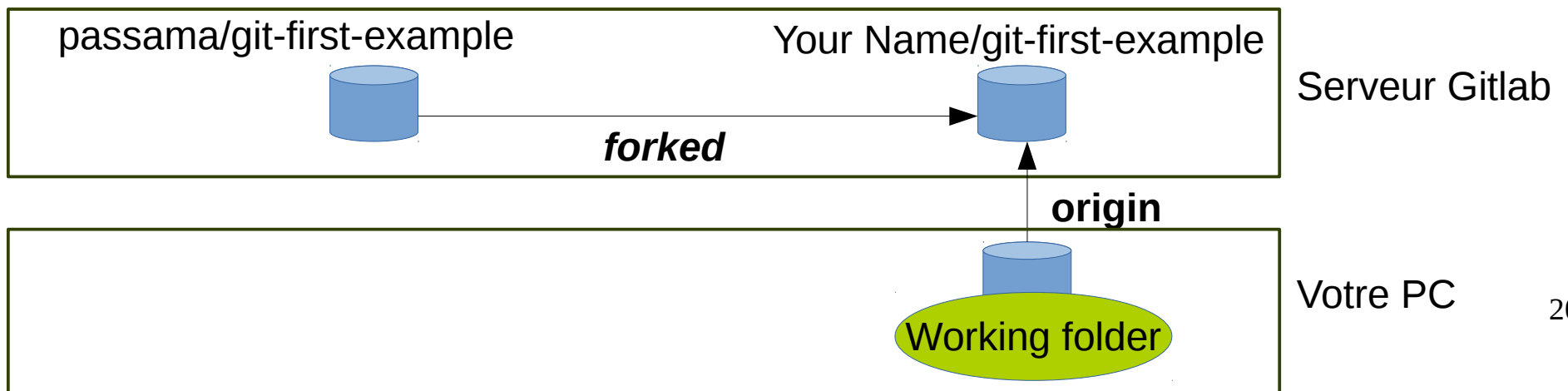
- Publier les modifications sur le dépôt serveur (push)
 - `git push origin dev`
Échec !!!
 - C'est normal : vous n'avez juste pas les droits de publier sur ce dépôt !
- Solution: ***Fourcher (fork) le dépôt serveur dans votre espace de travail personnel***
 - Faites le fork depuis GitLab
 - Comme vous serez le propriétaire du nouveau dépôt, **vous pourrez publier sur toutes les branches.**
 - Copier l'adresse du dépôt ainsi fourché.

Premier projet

- Changeons l'origine de votre dépôt local
- Ancienne architecture



- Nouvelle architecture



Premier projet

- Pour changer l'origine de votre dépôt local
 - `git remote set-url origin <address of the clone repository>`
- Vérifions le changement
 - `git remote -v`

Premier projet

- Publions à nouveau nos modifications sur le dépôt serveur fourché

- `git push origin dev`

Maintenant ça marche !

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 334 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To `git@gite.lirmm.fr:passama/git-first-example.git`

`0ca3e2d..321b394 dev -> dev`

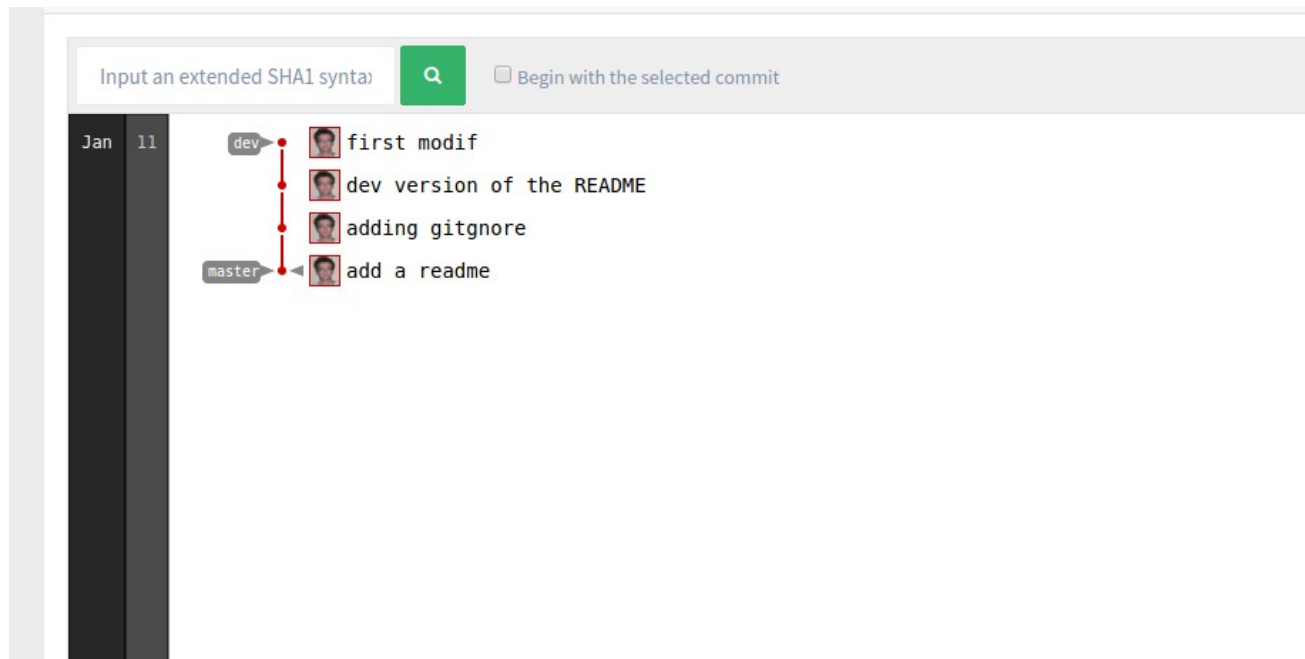
Dépôt distant
cible

Branche locale (source)

Branche distante (mise à jour)

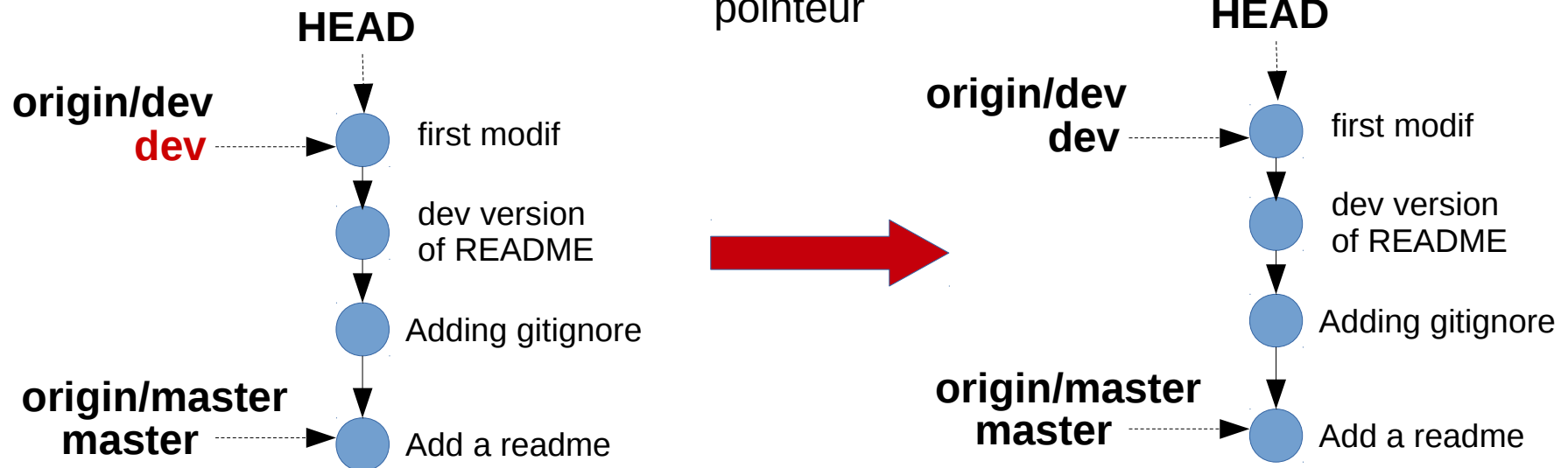
Premier projet

- Vérifier les modifications dans GitLab
 - Menu “repository” > onglet “graph”
- Vous devez voir quelque chose comme ça:



Premier projet

- Créons une nouvelle branche
 - `git checkout -b new-feature`
 - `git branch #current branch has changed`
- Graphiquement



Premier projet

- Créons du nouveau contenu
 - `mkdir dir && gedit dir/newfile`
 - Mettre un bon paragraphe de texte dans `newfile`
 - `gedit dir/otherfile`
 - Mettre un peu de texte dans `otherfile`

- Vérifions le status:

- `git status`

Sur la branche new-feature

Fichiers non suivis:

(utilisez `"git add <fichier>..."` pour inclure dans ce qui sera validé)

dir/

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez `"git add"` pour les suivre)

Premier projet

- On veut enregistrer les modifications **en 2 fois (2 commits)**

- `git add dir/newfile`

- `git status`

Sur la branche new-feature

Modifications qui seront validées :

(utilisez "git reset HEAD <fichier>..." pour désindexer)

nouveau fichier: dir/newfile

Fichiers non suivis:

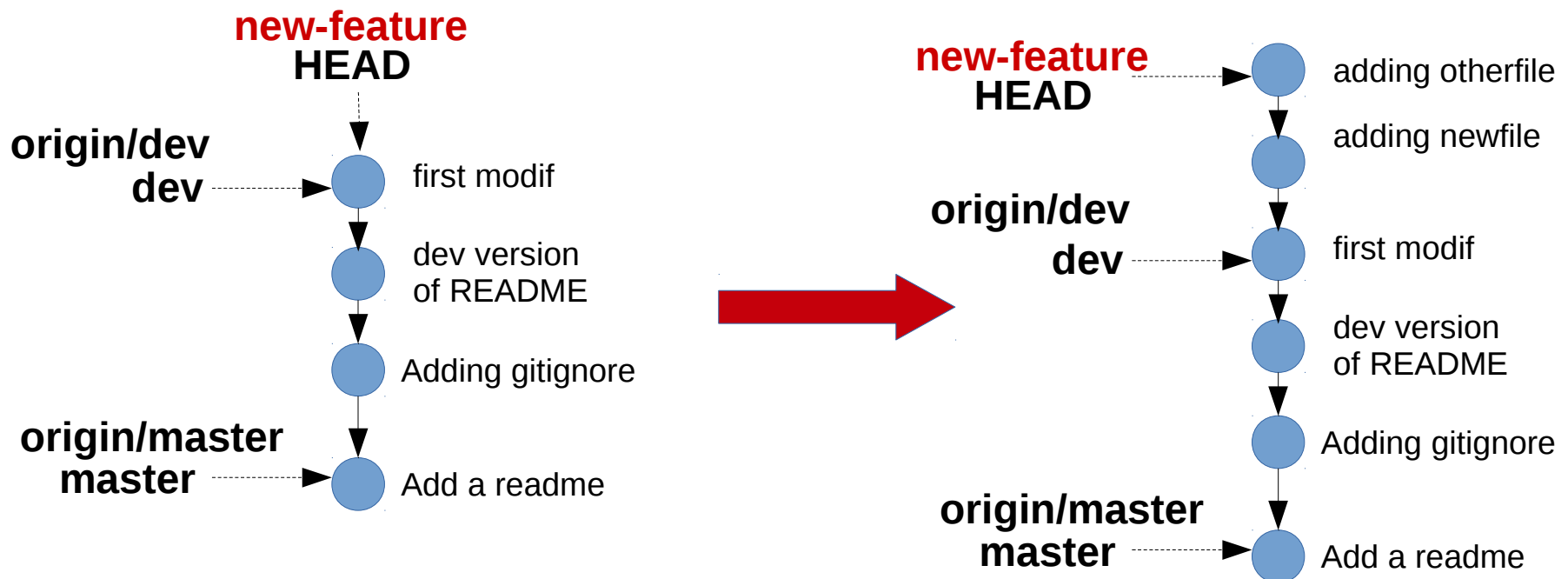
(utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

dir/otherfile

- `git commit -m "adding newfile"`

Premier projet

- Deuxième étape
 - `git add dir/otherfile`
 - `git commit -m "adding otherfile"`

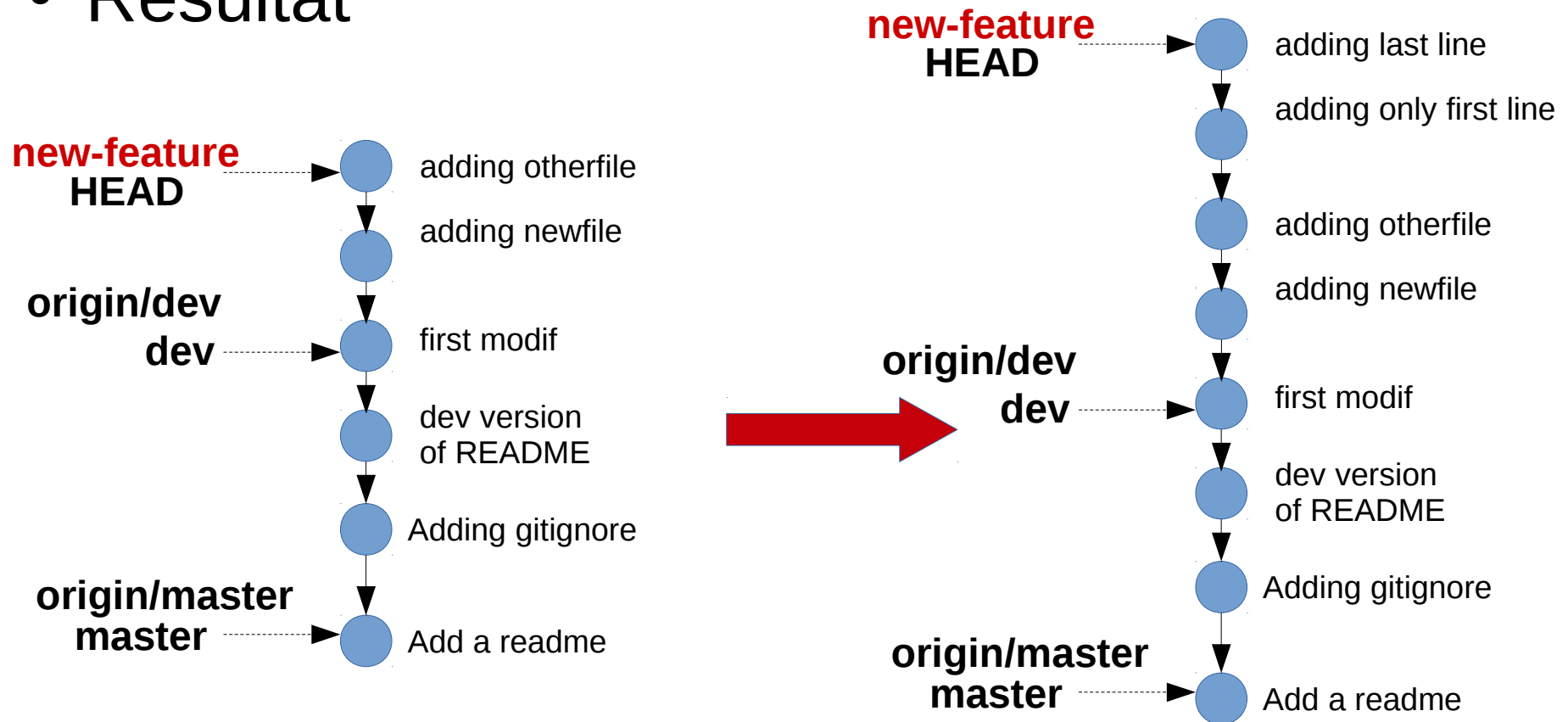


Premier projet

- Créer 2 commits à partir de 2 modifications
 - Éditer `newfile` et rajouter une nouvelle ligne au début, une nouvelle ligne à la fin
 - `git add -p` #sélectionne les modifs une à une
 - Taper 'y' pour sélectionner la modification de la 1ère ligne
 - Taper 'n' pour ne pas sélectionner la modification de la 2ème ligne
 - `git commit -m "adding only first line"`
 - `git add -p`
 - Taper 'y' pour sélectionner la modification de la 2ème ligne
 - `git commit -m "adding last line"`

Premier projet

- Résultat



Bonne pratique: Utiliser `git add -p` par défaut pour vérifier les modifications que vous allez valider

Premier projet


- Annuler une suite de commits
 - Nous voulons annuler les 2 derniers commits...
 - **git log**
 - Copiez l'identifiant haché du commit précédant les 2 commits à annuler
 - **git reset <hash-ID>**

Modifications non indexées après reset :

M dir/newfile
 - **git status**

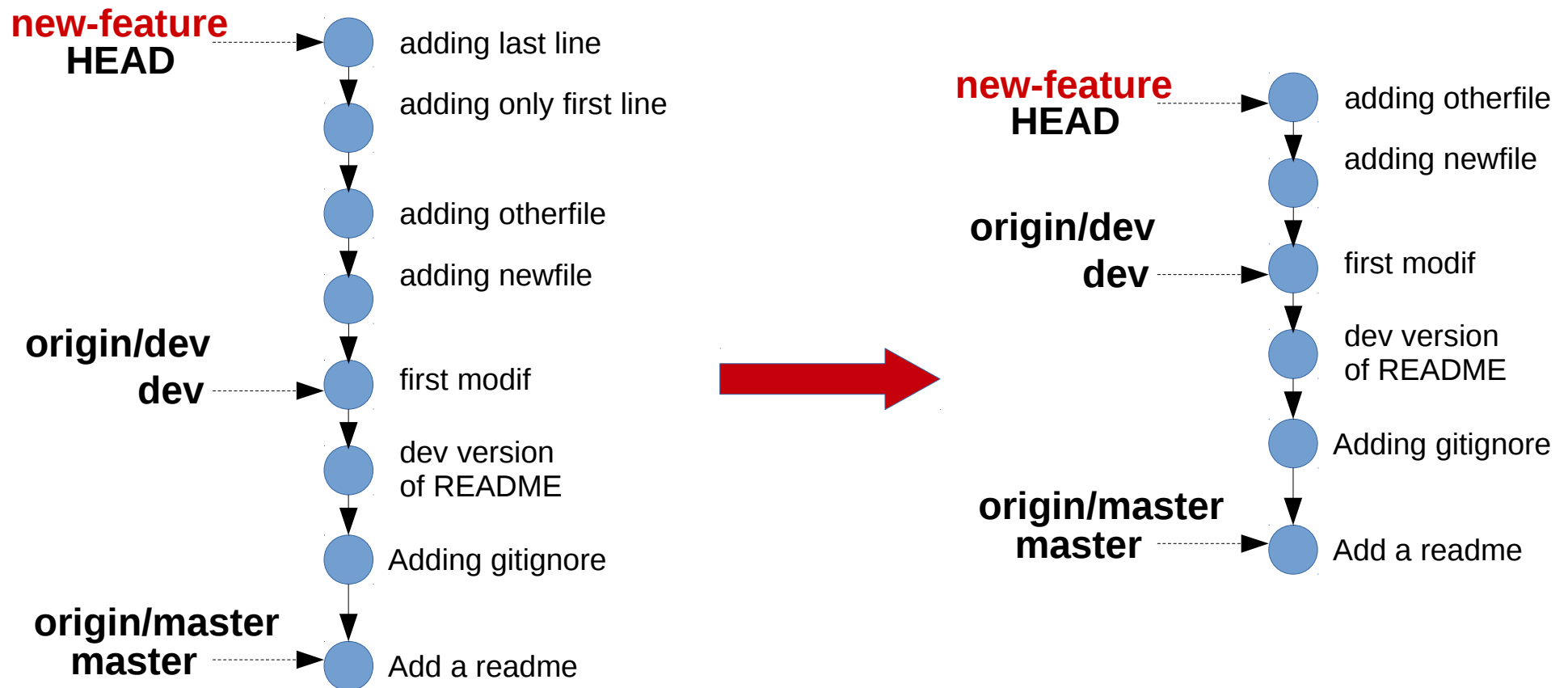
Modifications qui ne seront pas validées :

modifié: dir/newfile


 - Les modifications des commits annulés **sont à nouveau dans votre espace de travail**
-  **ATTENTION: Ne jamais utiliser git reset sur du contenu déjà publié (seulement sur des commits locaux non 'poussés')**

Premier projet

- Résultat



Premier projet

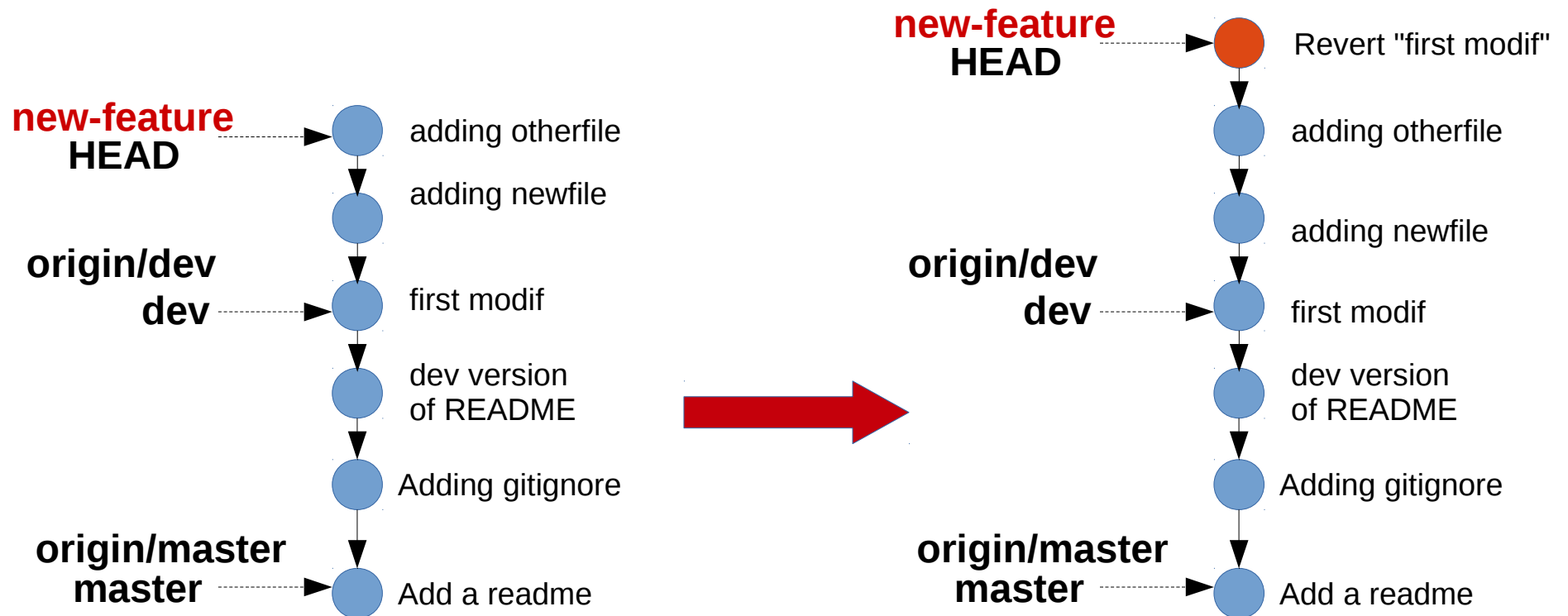
- Se débarrasser de ces modifications
 - Comment supprimer ces modifs de mon espace de travail ? 2 solutions:
 - `git reset --hard <SHA-1-ID> #use --hard in previous command`
 - Nettoyage définitif 
 - `git stash` (or `git stash save`)
 - Les modifications dans l'espace de travail **sont mises dans un commit temporaire et enlevées de l'espace de travail**. Vous pouvez alors soit:
 - Réappliquer les changements à l'espace de travail:
 - `git stash pop`
 - Oublier définitivement toutes les modifications planquées (stashed)
 - `git stash clear`

Premier projet

- Retourner à un commit
 - Finalement nous voulons défaire les modifications enregistrées dans “**first modif**”.
 - **git log**
 - Copier l’identifiant haché du commit auquel vous voulez retourner.
 - **git revert <hash-ID>**
 - Un nouveau commit est généré !

Premier projet

- Résultat



➔ **Bonne pratique:** utiliser git revert par défaut car c'est moins dangereux (marche sur des commits publiés)

Premier projet

- La nouvelle fonctionnalité est finie, nous voulons mettre à jour `dev` avec celle-ci.

- `git checkout dev #go to dev branch`
- `git merge new-feature`

Mise à jour 321b394..2a77b12

Fast-forward

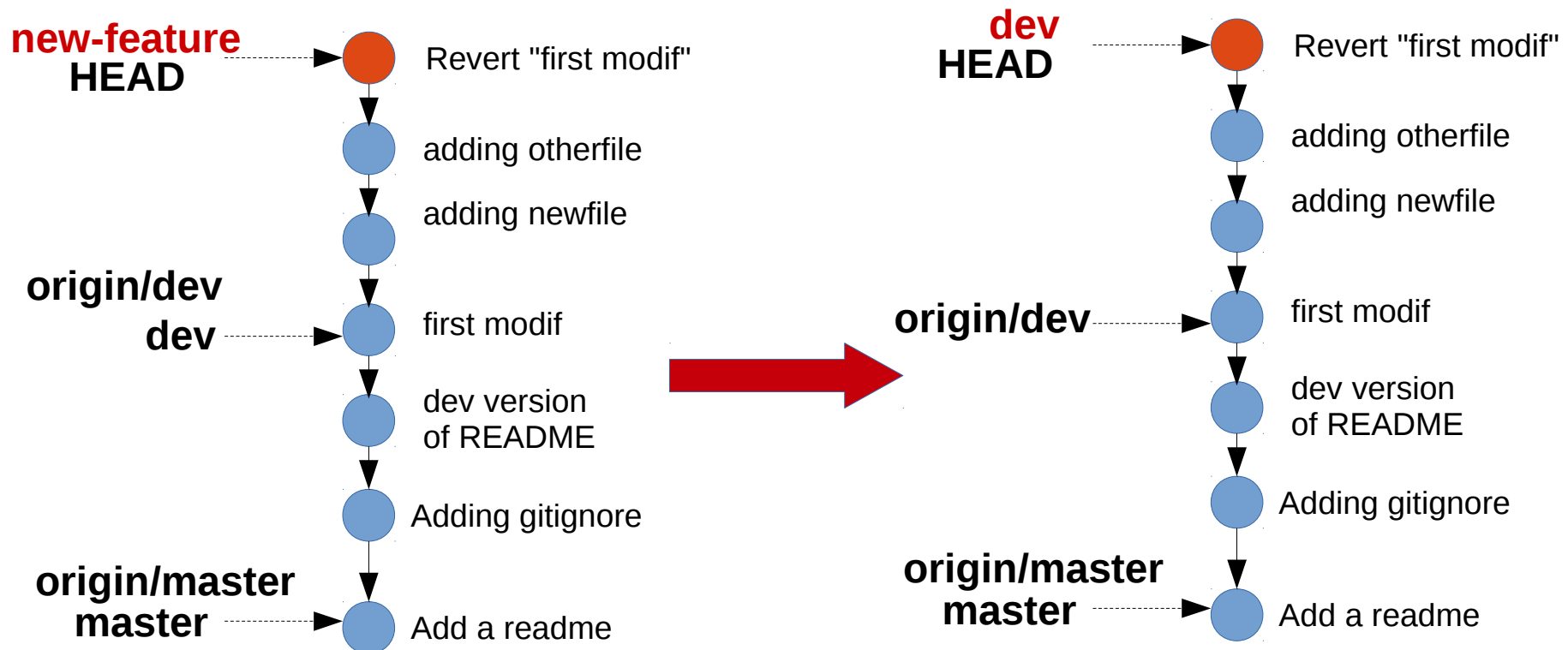
```
README.md      | 1 -  
dir/newfile    | 7 +++++++  
dir/otherfile  | 6 +++++++  
3 files changed, 13 insertions(+), 1 deletion(-)  
create mode 100644 dir/newfile  
create mode 100644 dir/otherfile
```

Résume toutes
les modifications
depuis le dernier
commit de *dev*

- Il reste à supprimer la branche `new-feature` (qui n'est plus utile)
 - `git branch -D new-feature`

Premier projet

- Result

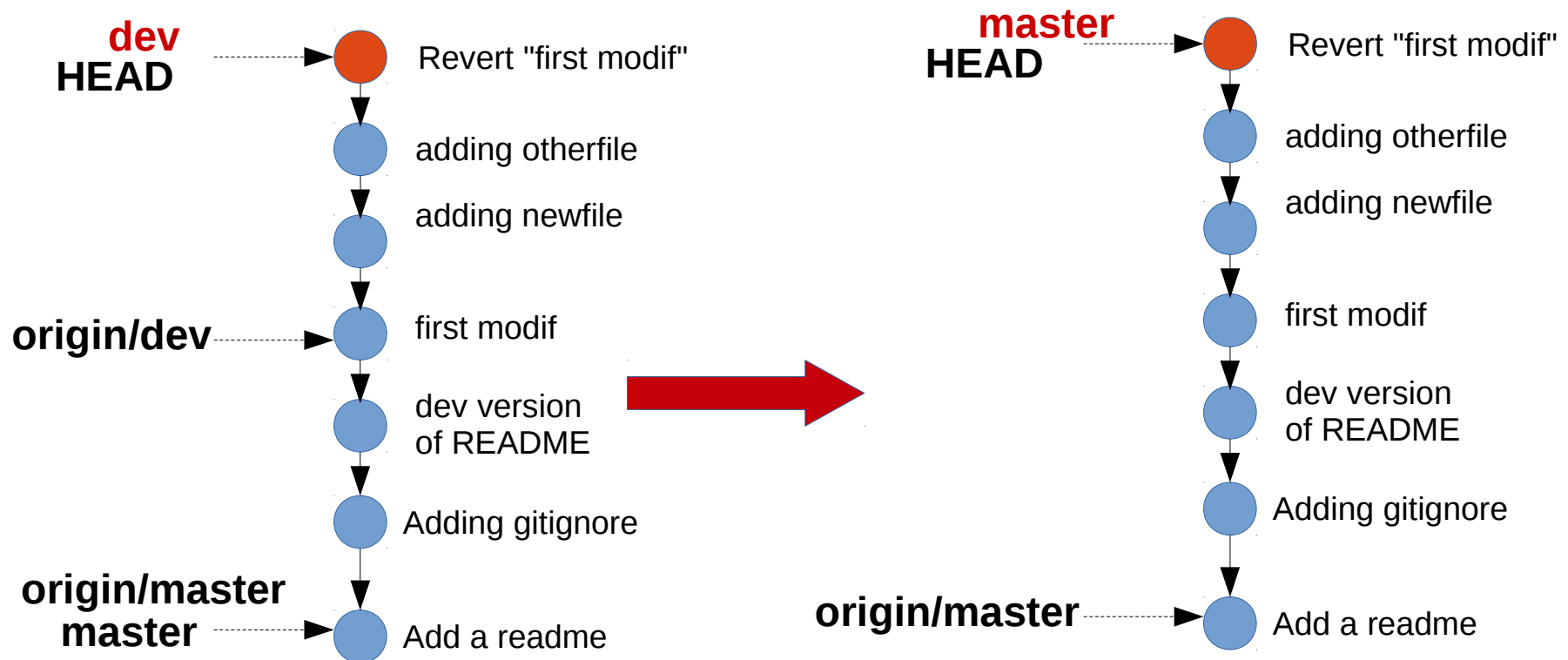


Premier projet

- Le développement est fini, nous n'avons plus besoin de `dev`.
 - `git checkout master` #go to master branch
 - `git merge dev`
 - Nous voulons supprimer `dev` sur le PC local et sur le serveur
 - Supprime la branche locale
 - `git branch -D dev`
 - Supprime la branche distante
 - `git push --delete origin dev`
- To `git@gite.lirmm.fr:passama/git-first-example.git`
- [deleted] dev

Premier projet

- Résultat

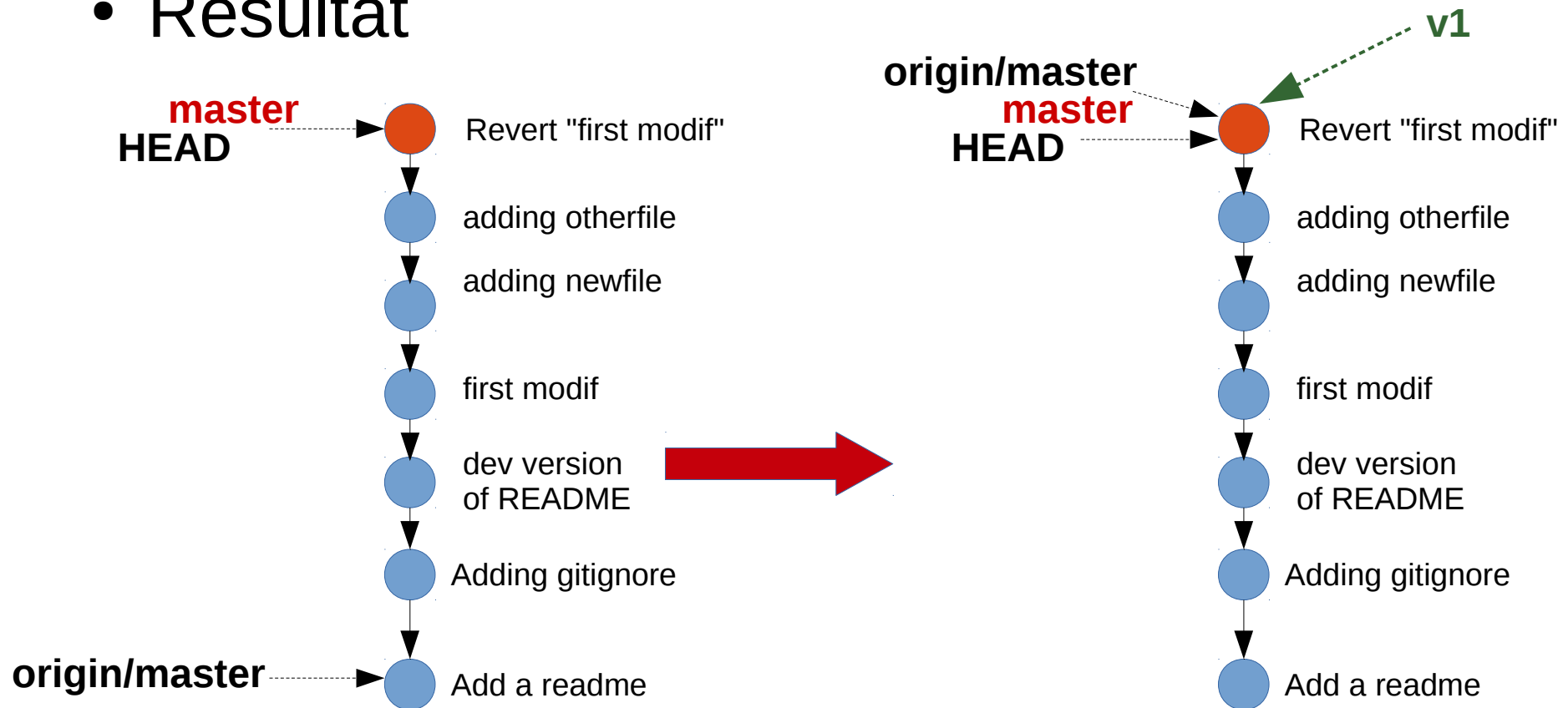


Premier projet

- Sauvegardons l'état courant du dépôt
 - `git tag -a v1 -m "version1"`
- Mettons à jour le dépôt sur le serveur
 - `git push origin master`
 - `git push origin v1`
- Vous pouvez à tout moment revenir à cet état
 - `git checkout v1`

Premier projet


- Résultat



Plan

- Installation
- **Tutoriel**
 - Un premier projet
 - **Travail collaboratif**
 - Conseils et astuces

Travail collaboratif

- Créer dans Gitlab un groupe de 3-5 personnes
 - Un groupe est un ensemble de projets reliés
 - Un groupe définit un nouvel espace de travail pour des projets
 - Un groupe définit un ensemble de développeurs travaillant sur ces projets.
-  **Les noms des groupes doivent être uniques sur le serveur**

Travail collaboratif

- Vous pouvez ajouter des membres à votre groupe

Défini le rôle des nouveaux membres

Sélectionne un nouveau membre

The screenshot shows the 'common-docs - Members' page in GitLab. It features a search bar at the top right. Below it, the 'Add new user to group' section includes a 'People' search input and a 'Group Access' dropdown menu currently set to 'Guest'. A green 'ADD USERS TO GROUP' button is positioned below these fields. The lower section, 'common-docs group members (3)', lists three members: Thierry GIL (gil), Joel MAIZI (maizi), and Robin Passama (passama). Each member entry shows their role (Master, Owner, Owner) and a 'LEAVE' button. Annotations with black dots and lines point to specific elements: one dot on the 'Search for a user' input is labeled 'Sélectionne un nouveau membre'; another dot on the 'Group Access' dropdown is labeled 'Défini le rôle des nouveaux membres'; a third dot on the member list is labeled 'Membres déjà enregistrés'; and a fourth dot on the 'LEAVE' button is labeled 'Rôles des membres'.

Membres déjà enregistrés

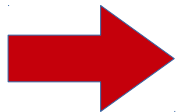
Rôles des membres

Travail collaboratif

- Comprendre les permissions reliées aux rôles
 - Rôles disponibles :
 - Guest: ne peut pas pull/clone le dépôt, peut seulement créer des “issues” (tickets).
 - Reporter: Guest + peut pull/clone le dépôt
 - Developer: Reporter + peut contribuer (peut pousser sur des branches non protégées, créer et gérer des merge request, écrire dans le wiki, etc.)
 - Maintainer: Developer + gestion de l'équipe, gestion de la protection des branches, peut pousser dans les branches protégées, peut créer des projets associés au groupe).
 - Owner: Master + gestion de la configuration du projet (création, renommage, suppression, visibilité, etc.), gestion des membres du groupe.
 - **Un rôle dans un groupe implique un rôle équivalent ou supérieur dans les projets du groupe.**

Travail collaboratif

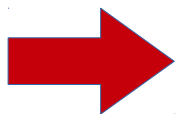
- Créez un projet `test-git` dans le groupe (pour le Master ou Owner du groupe)
 - Les membres du projet doivent être au moins **Developer** (peut publier (=push)).
 - Vous pouvez inviter dans votre projet des utilisateurs qui ne sont pas dans le groupe.
 - Vous pouvez toujours augmenter les permissions des membres du groupe.



Cliquer sur “new project” sur la page de groupe.

Travail collaboratif

- Le **Owner** de `test-git` doit initialiser le projet :
 - Localement, dans un terminal :
 - `cd <somewhere>`
 - `mkdir test-git`
 - `cd test-git`
 - editez `README.md` et un fichier `.gitignore` (pour ignorer les fichiers temporaires)
 - `git init` #transform an existing folder into a git repository
 - `git add --all`
 - `git commit -m "first commit"`
 - `git remote add origin <address of the project created in Gitlab>`
 - `git push origin master`
- Votre projet est initialisé dans Gitlab



Bonne pratique: ayez toujours un `README.md` à la création de projet (utilisez la syntaxe markdown) pour générer des pages d'accueil simple.

Travail collaboratif

- Pour les autres membres du groupe
 - Localement, dans un terminal :
 - `cd <somewhere>`
 - `git clone <address of the project created in Gitlab>`
- **Vous êtes prêt pour du travail collaboratif**

Travail collaboratif

- Le **Owner** ou le **Master** crée un fichier `file1.c`, et écrit dedans:

```
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

- Puis il valide (commit) et publie (push) ses modifications :

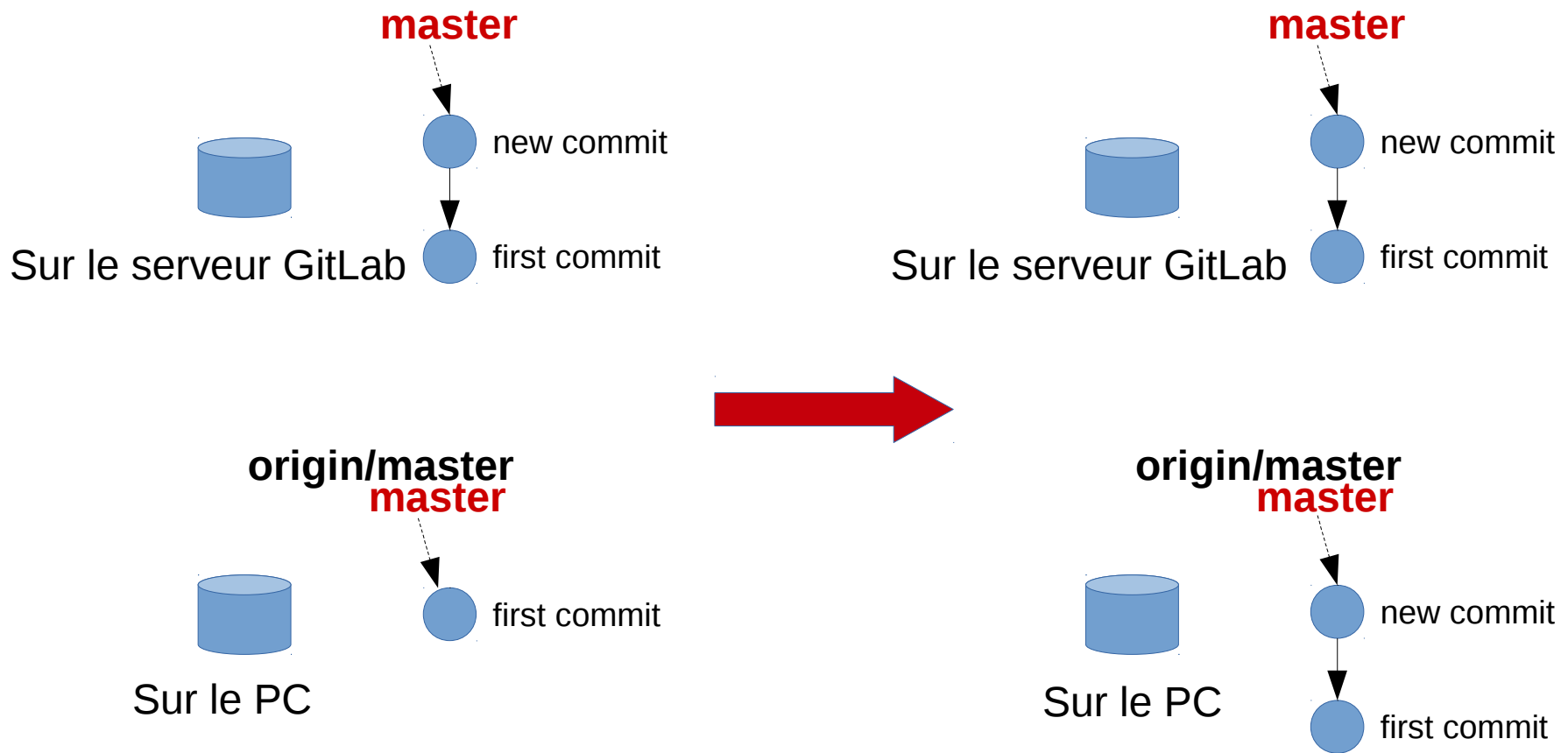
```
- git add file1.cpp
- git commit -m "adding file1"
- git push origin master
```


Travail collaboratif

- Maintenant les autres membres mettent à jour leurs dépôts locaux:
 - `git pull origin master`
- Rappel : `pull` fait 2 choses :
 - un `fetch` sur le dépôt (récupère les nouvelles modifications du dépôt serveur).
 - Une fusion (`merge`) de la branche `origin/master` dans la branche locale `master`.

Travail collaboratif

- Effet du `pull`

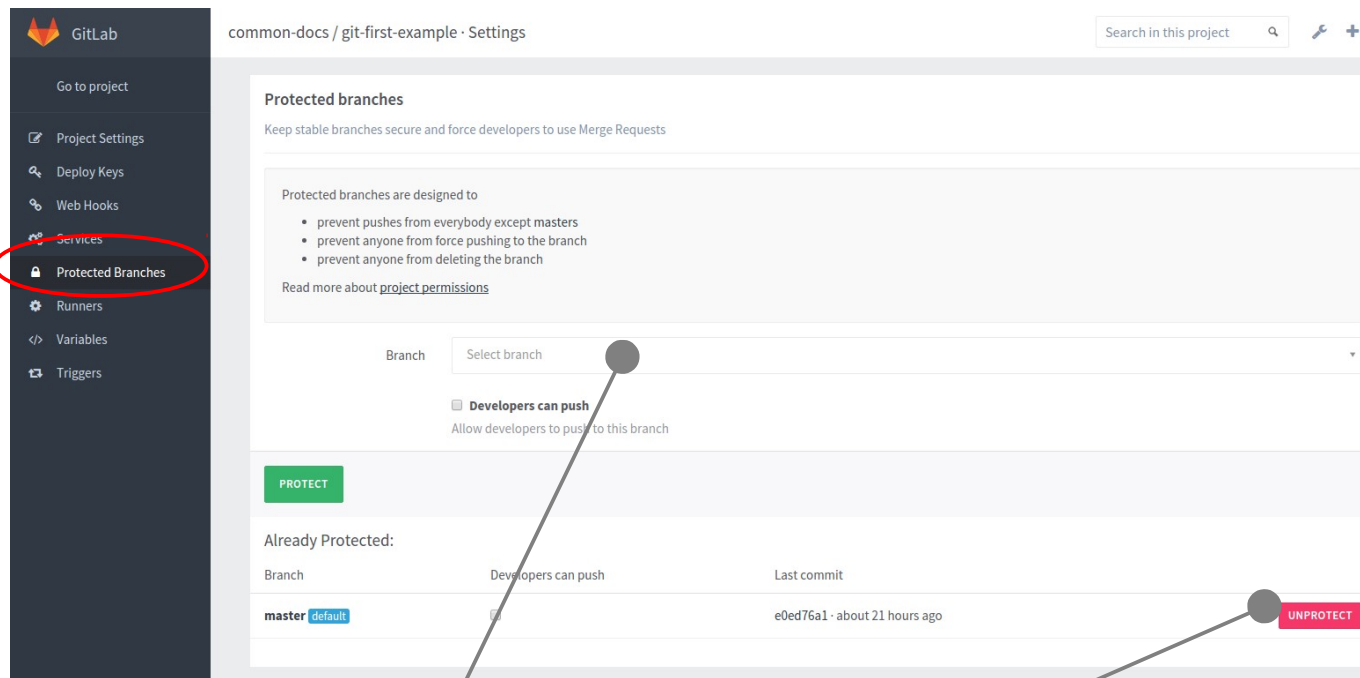


Travail collaboratif

- Les utilisateurs qui ne sont pas **Master** ou **Owner** modifient `file1.cpp` dans leur dépôt local.
- Valider (commit) et publier (push) les modifications sur master
 - `git push origin master`
ÉCHEC
- C'est normal puisque `master` est **protégée** par défaut
 - Seuls **Masters** et **Owners** peuvent pousser sur les branches protégées (protection par défaut).
 - Pourquoi ? Prévenir la suppression de branche et de mauvaises publications par les développeurs

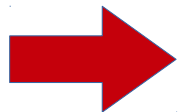
Travail collaboratif

- Gestion des branches protégées dans Gitlab
 - Aller dans settings > repository > “protected branches”



Pour protéger une branche

Pour dé-protéger une branche



Bonne pratique: garder la branche `master` protégée pour ne pas permettre aux développeurs de publier (push)

Travail collaboratif

- Solution: les développeurs créent une autre branche et proposent une “merge request”
 - Créer une nouvelle branche sur le serveur
 - `git checkout -b <my-branch-name> #local`
 - `git push origin <my-branch-name> #serveur`
 - Proposer une “merge request”
 - Créer une nouvelle “merge request” dans Gitlab avec
 - `<my-branch-name>` comme source
 - `master` comme cible

Travail collaboratif

- Owner et Master peuvent gérer les “merge request” dans GitLab
 - Dans le menu “merge requests” du projet, vérifier les modifications,
 - Si c’est OK, acceptez le “merge request”.
 - En cas de conflits, il faut les résoudre “à la main” (i.e. dans votre dépôt local):
 - `git checkout master #en cas de conflits`
 - `git pull origin master #maj master`
 - `git pull origin <branch name> #fusion`
 - Devrait se plaindre d’un conflit

Travail collaboratif

- Résoudre un conflit

- Pour obtenir des informations sur le conflit

- **git status**

- # On branch master

- # You have unmerged paths.

- # (fix conflicts and run "git commit")

- #

- # Unmerged paths:

- # (use "git add ..." to mark resolution)

- #

- # both modified: file1.cpp

Fichiers contenant des conflits

Travail collaboratif

- Résoudre le conflit
 - En ouvrant ces fichiers, vous devriez voir des choses comme ça

```
the number of planets are
```

```
<<<<<< HEAD
```

```
nine
```

Ce que votre branche
courante contient

```
=====
```

```
eight
```

```
>>>>>> <the branch name>
```

Ce que la branche
fusionnée contient

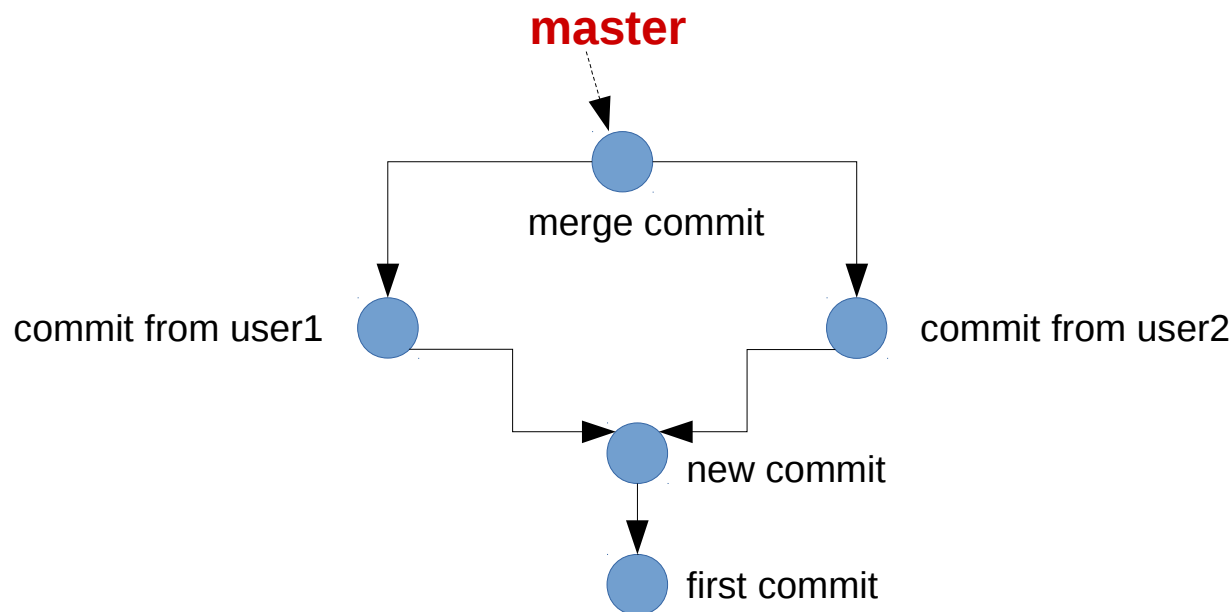
Travail collaboratif

- Résoudre le conflit
 - Résoudre = choisir une alternative (or réécrire le tout) + supprimer les commentaires spécifiques <<< ou >>>

```
the number of planets are  
eight
```
 - Faire un commit de résolution:
 - `git commit -am "conflict on planets resolved" #add --all and commit in one step is possible`
 - Mettre à jour la branche master du serveur
 - `git push origin master`

Travail collaboratif

- Le dépôt après la fusion (conflit ou non)



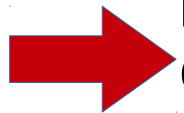
- Si il n'y a pas de conflits, le commit de fusion est généré automatiquement

Travail collaboratif

- Généralement, les **Developers** doivent résoudre les conflits eux-même

Créer une branche “bac à sable” entre les contributeurs, e.g. `dev`, `integration...`

- `git checkout -b dev`
- `git push origin dev`
- La branche `master` n’est mise à jour (par fusion) que quand l’état de la branche `dev` est “stable”.



Bonne pratique: Protéger la branche `dev` mais autoriser les développeurs à pousser (évite que la branche ne soit supprimée)

Travail collaboratif

- Tous les utilisateurs
 - Récupèrent la branche `dev`
 - `git fetch origin #update repository`
 - `git checkout dev #local dev branch is automatically created`
 - Écrivent leur code et commit dans `dev`
 - Mettent à jour avec un pull de `origin dev`
 - Si nécessaire, résolvent les conflits sur leur dépôt local
 - Puis poussent sur `origin dev`
 - Etc.

Plan

- Installation
- **Tutoriel**
 - Un premier projet
 - Travail collaboratif
 - **Conseils et astuces**

Conseils et astuces

- Configurer la visibilité de votre projet
 - Privée (symbole = verrou).
 - Seuls les membres du projet peuvent cloner/fourcher.
 - Public (symbole = planète).
 - N'importe qui peut cloner !
 - Pour donner une audience réelle à un projet plutôt utiliser un service comme github.com ou gitlab.com.
 - Interne (symbole =bouclier).
 - n'importe quelle personne enregistrée dans gitlab peut cloner.
 - Bonne approche pour permettre à d'autres de contribuer à vos projet suivant l'approche **fork/merge requests**.

Conseils et astuces

- Dans Gitlab, utiliser les **issues** et les **code snippets** pour communiquer entre vous sur les bugs, améliorations, etc.
 - **Issues** : meilleur moyen de garder en mémoire les « choses à faire ».
 - Utiliser **les labels** sur les issues pour clairement identifier la portée de l'issue (bugs, documentation, etc.)
 - Utiliser les **code snippet** pour partager des exemples of code, reporter de long messages d'erreurs, etc.

Conseils et astuces

- Utiliser `git-svn` pour porter des projets SVN dans git.
- Importer un dépôt SVN et le transformer en dépôt git

```
git svn clone <address> -s
```



Cette opération peut prendre beaucoup de temps !

- Créer un le projet correspondant dans gitlab puis

```
git remote rename origin svn-server
```

```
git remote add origin <gitlab project address>
```

- Pusher toutes les branches et tags vers ce dépôt git

```
git push origin --all #pushing all branches
```

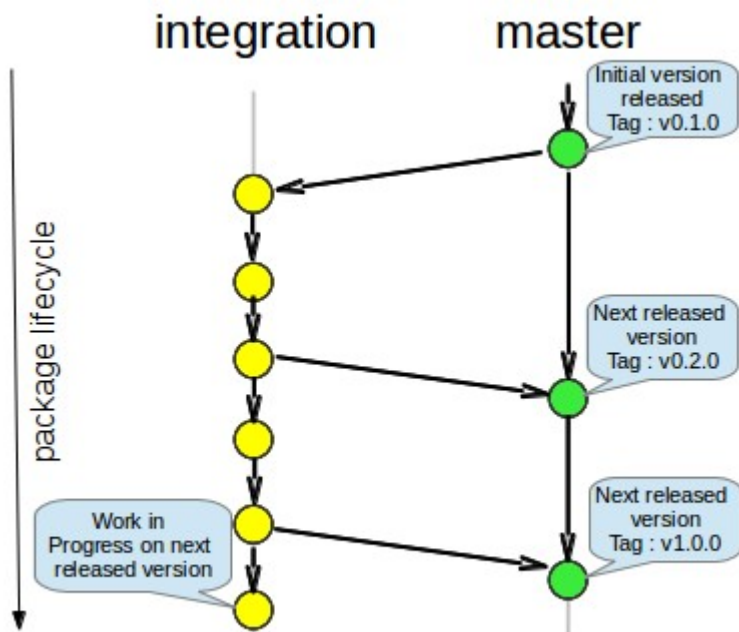
```
git push origin --tags #pushing all tags
```


Conseils et astuces

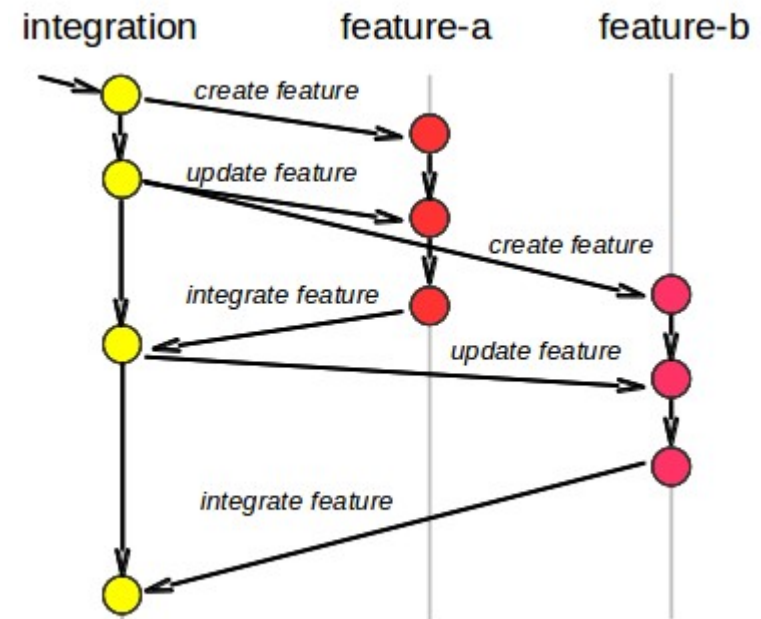
- Ignorer certains fichiers avec **gitignore**
 - Toujours créer un fichier `.gitignore` à la racine de votre projet.
 - Ajouter des règles pour supprimer les fichiers temporaires générés par vos outils de développement
règle : `*~`
 - Pour organiser la structure de votre projet
 - Ajouter un fichier `.gitignore` pour chaque répertoire vide qui doit apparaître au moment du clone (exemple `build/`, `bin/`, `lib/`).
Règle (ne suivre aucun fichier) : `*`
 - Ces dossiers existent dans le dépôt mais pas leur contenu (excepté le fichier `.gitignore`) !

Conseils et astuces

- Un modèle général de gestion des branches
 - **Integration:** branche « protégée » et **cocher** “Developers can push”
 - **Master:** branche « protégée » et **décocher** “Developers can push”



branches permanentes (protégées)



branches temporaires (non protégées)

Conseils et astuces

- Utiliser `gitg` pour voir l'état d'un dépôt local (sur votre station de travail)

