Introduction au gestionnaire de versions GIT

Olivier Caron

Polytech Lille
Avenue Paul Langevin Cité Scientifique Lille 1
59655 Villeneuve d'Ascq cedex

http://ocaron.plil.fr
Olivier.Caron@polytech-lille.fr

Références bibliographiques

- Livre ”Pro-Git” De Scott Chacon and Ben Straub
  `https://git-scm.com/book/fr/v2`

Références bibliographiques

- Livre ”Pro-Git” De Scott Chacon and Ben Straub
  `https://git-scm.com/book/fr/v2`
- Tutoriel GIT ”Pour arrêter de galérer avec Git”
  `https://www.miximum.fr/blog/enfin-comprendre-git/`

Références bibliographiques

- Livre ”Pro-Git” De Scott Chacon and Ben Straub
  `https://git-scm.com/book/fr/v2`
- Tutoriel GIT ”Pour arrêter de galérer avec Git”
  `https://www.miximum.fr/blog/enfin-comprendre-git/`
- Présentation GIT ”Les bases de GIT”
  `https://fr.slideshare.net/PierreSudron/diapo-git`

Références bibliographiques

- Livre ”Pro-Git” De Scott Chacon and Ben Straub
  `https://git-scm.com/book/fr/v2`
- Tutoriel GIT ”Pour arrêter de galérer avec Git”
  `https://www.miximum.fr/blog/enfin-comprendre-git/`
- Présentation GIT ”Les bases de GIT”
  `https://fr.slideshare.net/PierreSudron/diapo-git`
- Cours de Walter Rudametkin
  `https://rudametw.github.io/teaching`

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers
- De multiples itérations sont nécessaires.

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers
- De multiples itérations sont nécessaires.
- A certains moments, on peut identifier des versions et/ou variantes du logiciel.

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers
- De multiples itérations sont nécessaires.
- A certains moments, on peut identifier des versions et/ou variantes du logiciel.
- Les erreurs sont possibles, revenir en arrière est parfois nécessaire.

Problématique (1/2)

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers
- De multiples itérations sont nécessaires.
- A certains moments, on peut identifier des versions et/ou variantes du logiciel.
- Les erreurs sont possibles, revenir en arrière est parfois nécessaire.
- Un projet peut se faire à plusieurs, les développeurs peuvent travailler sur les mêmes fichiers (conflits)

Problématique (1/2)

- Le développement logiciel est un processus sinueux
  ➔ notion de **branche** :

Qu’est ce qu’un logiciel de gestion de versions

Définition simple

Un gestionnaire de versions est un logiciel qui enregistre les évolutions d’un ensemble de fichiers au cours du temps de manière à ce qu’on puisse rappeler une version antérieure à tout moment

Qu'est ce qu'un logiciel de gestion de versions

Définition simple

Un gestionnaire de versions est un logiciel qui enregistre les évolutions d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure à tout moment

- Et, en plus :

Qu'est ce qu'un logiciel de gestion de versions

Définition simple

Un gestionnaire de versions est un logiciel qui enregistre les évolutions d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure à tout moment

- Et, en plus :
  - Faciliter la détection et correction d'erreurs

Qu'est ce qu'un logiciel de gestion de versions

Définition simple

Un gestionnaire de versions est un logiciel qui enregistre les évolutions d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure à tout moment

- Et, en plus :
  - Faciliter la détection et correction d'erreurs
  - Peut gérer plusieurs utilisateurs (permissions, savoir qui a fait quoi, gestion des conflits,. . .)

Le gestionnaire de versions GIT

- Créé en 2005 par Linus Torvalds (également créateur de linux) pour gérer le noyau linux.

Le gestionnaire de versions GIT

- Créé en 2005 par Linus Torvalds (également créateur de linux) pour gérer le noyau linux.
- Complètement distribué, rapide

Le gestionnaire de versions GIT

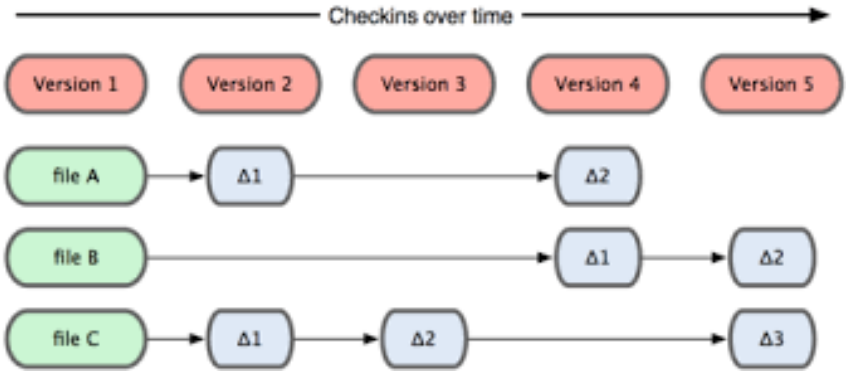
- Créé en 2005 par Linus Torvalds (également créateur de linux) pour gérer le noyau linux.
- Complètement distribué, rapide
- Fonctionne sans réseau **ET** avec réseau !

Le gestionnaire de versions GIT

- Créé en 2005 par Linus Torvalds (également créateur de linux) pour gérer le noyau linux.
- Complètement distribué, rapide
- Fonctionne sans réseau **ET** avec réseau !
- Capable de gérer des projets d'envergure et des milliers de branches en parallèle.

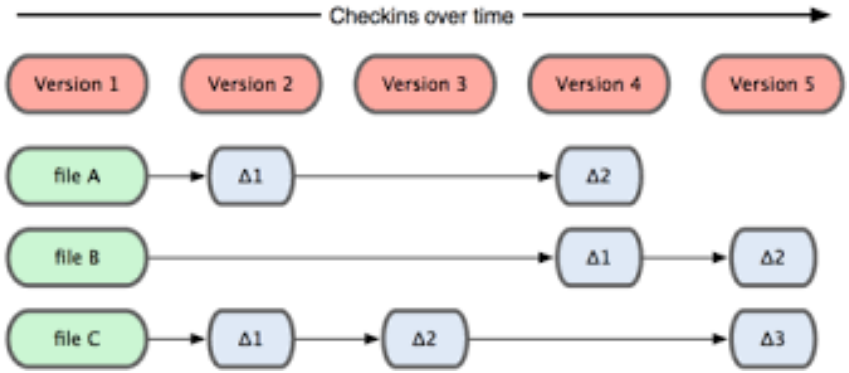
1ère originalité de GIT:des snapshots, pas des différences

- Autres systèmes de gestion de versions :

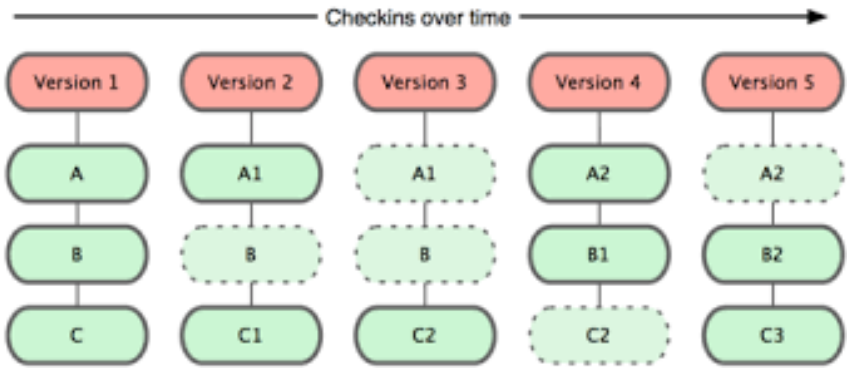


1ère originalité de GIT:des snapshots, pas des différences

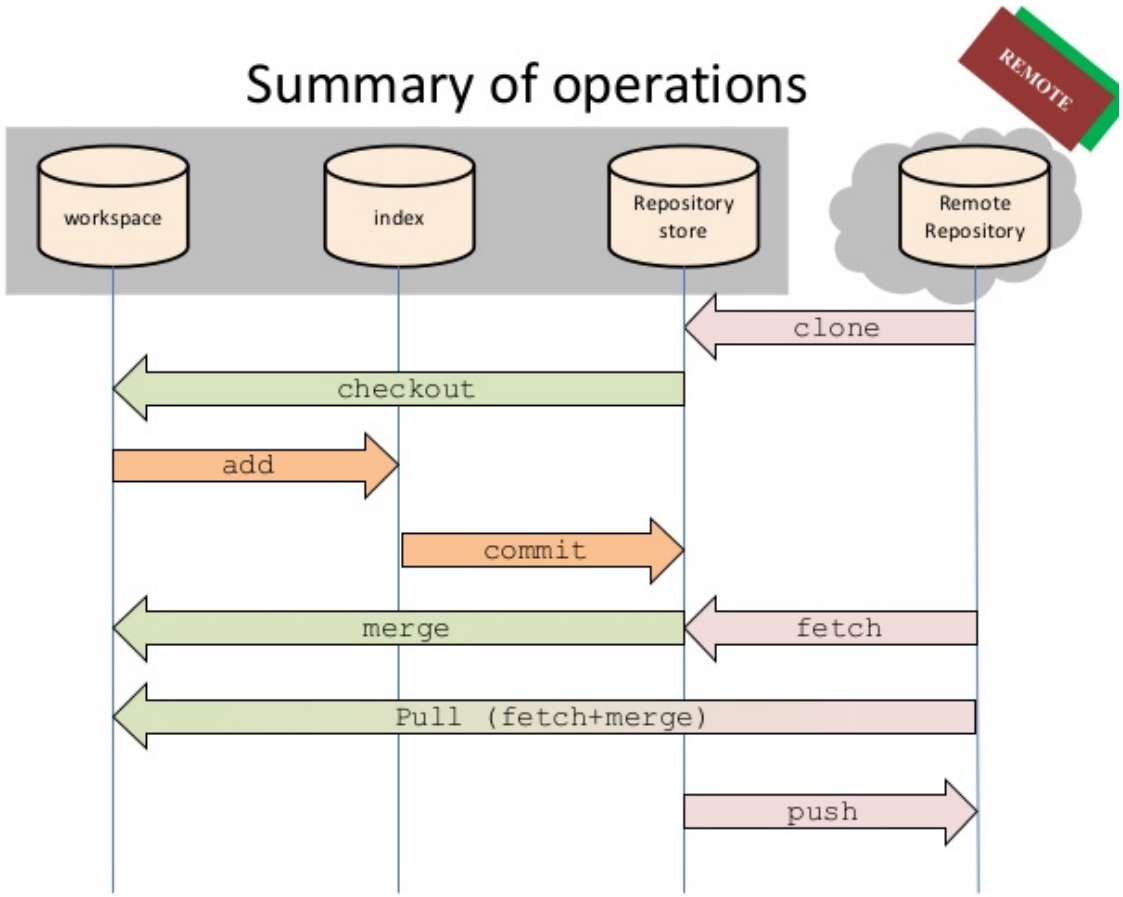
- Autres systèmes de gestion de versions :



- GIT:



2nde originalité de GIT: 4 espaces



- source : `https://www.slideshare.net/YoadSnapir/git-intro-42289838`

Let's go

- Première étape, configuration:

```
1  git config --global user.name "John Smith"
2  git config --global user.email "John.Smith@polytech-lille.net"
3  git config --global core.editor gedit
4  git config --global push.default simple
5  git config --global color.decorate full
6  git config --global merge.conflictstyle diff3
```

- A faire une seule fois: informations stockées dans `~/.gitconfig`
- Ligne 3 : choix de l'éditeur : kate, gedit, emacs, vi, ...
- Disposez d'un prompt adapté :

```
source ~ocaron/public/git/config_git
```

Initialisation : création d'un projet et premier "commit"

- Ligne 2 : création des espaces `index` et `dépôt local` : répertoire `.git`

```
1  mkdir demoCours
2  git init demoCours
3  cd demoCours
4  echo \*~ > .gitignore
5  echo -n bruce > rock.txt
6  git add rock.txt .gitignore
7  git commit -m "premier commit"
```

Figure: État du dépôt local

dépôt vide

Initialisation : création d'un projet et premier "commit"

```
1  mkdir demoCours
2  git init demoCours
3  cd demoCours
4  echo \*~ > .gitignore
5  echo -n bruce > rock.txt
6  git add rock.txt .gitignore
7  git commit -m "premier commit"
```

- Ligne 2 : création des espaces `index` et `dépôt local` : répertoire `.git`
- Ligne 4 : patterns des fichiers ignorés

Figure: État du dépôt local

dépôt vide

Initialisation : création d'un projet et premier "commit"

```
1  mkdir demoCours
2  git init demoCours
3  cd demoCours
4  echo \*~ > .gitignore
5  echo -n bruce > rock.txt
6  git add rock.txt .gitignore
7  git commit -m "premier commit"
```

- Ligne 2 : création des espaces `index` et `dépôt local` : répertoire `.git`
- Ligne 4 : patterns des fichiers ignorés
- Ligne 6 : ajout des fichiers dans l'espace index

Figure: État du dépôt local après commit

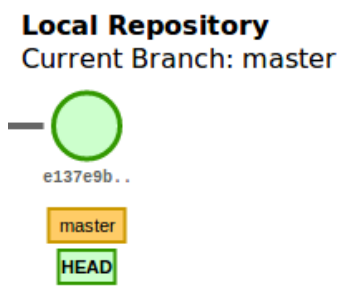
dépôt vide

Initialisation : création d'un projet et premier "commit"

```
1  mkdir demoCours
2  git init demoCours
3  cd demoCours
4  echo \*~ > .gitignore
5  echo -n bruce > rock.txt
6  git add rock.txt .gitignore
7  git commit -m "premier commit"
```

- Ligne 2 : création des espaces `index` et `dépôt local` : répertoire `.git`
- Ligne 4 : patterns des fichiers ignorés
- Ligne 6 : ajout des fichiers dans l'espace index
- Ligne 7 : première version dans dépôt local
  ➔ `commit_id`

Figure: État du dépôt local



Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.
- La référence `HEAD` désigne le commit courant : *"le répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par `HEAD`"*

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.
- La référence `HEAD` désigne le commit courant : *"le répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par `HEAD`"*
- Trois commandes très utiles :

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.
- La référence `HEAD` désigne le commit courant : *"le répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par `HEAD`"*
- Trois commandes très utiles :

  git status permet de découvrir les différences entre les espaces : fichiers non indexés, commit non fait.

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.
- La référence `HEAD` désigne le commit courant : *"le répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par `HEAD`"*
- Trois commandes très utiles :

  git status permet de découvrir les différences entre les espaces : fichiers non indexés, commit non fait.

  git log –oneline fournit l'historique des commit

Dénominations et commandes GIT

- La branche initiale, créée par défaut, s'intitule la branche `master`
- En utilisant le prompt git, on sait dans quelle branche l'espace de travail est associé.
- La référence `HEAD` désigne le commit courant : *"le répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par `HEAD`"*
- Trois commandes très utiles :

  git status permet de découvrir les différences entre les espaces : fichiers non indexés, commit non fait.

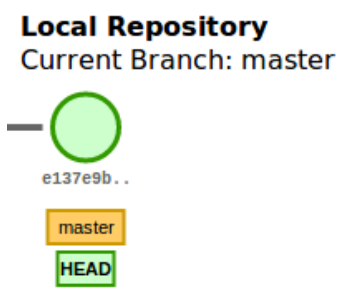
  git log –oneline fournit l'historique des commit

  git show commit_id décrit un commit particulier

Continuons à développer

```
1  echo -n ,mick >> rock.txt
2  git add rock.txt
3  git commit -m "avec mick"
4  echo -n ,bono >> rock.txt
5  git add rock.txt
6  git commit -m "avec bono"
```

Figure: État du dépôt local avant commit ligne 3

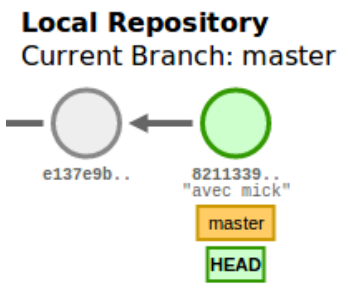


Continuons à développer

```
1  echo -n ,mick >> rock.txt
2  git add rock.txt
3  git commit -m "avec mick"
4  echo -n ,bono >> rock.txt
5  git add rock.txt
6  git commit -m "avec bono"
```

- Lignes 1-3 : évolution du fichier, second commit

Figure: État du dépôt local après commit ligne 3

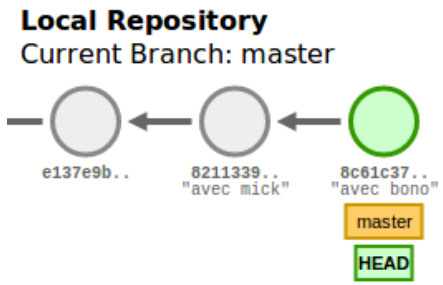


Continuons à développer

```
1  echo -n ,mick >> rock.txt
2  git add rock.txt
3  git commit -m "avec mick"
4  echo -n ,bono >> rock.txt
5  git add rock.txt
6  git commit -m "avec bono"
```

- Lignes 1-3 : évolution du fichier, second commit
- Lignes 3-6 : évolution du fichier, troisième commit

Figure: État du dépôt local après commit ligne 6

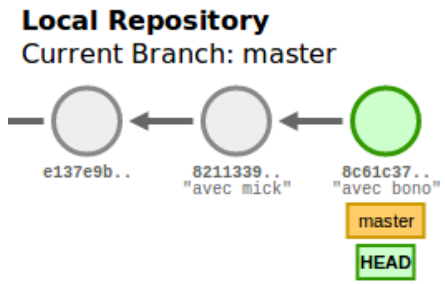


Back to the future (”Nom de Zeus”)

```
1  cat rock.txt
2  bruce,mick,bono
3  git log --oneline
4  8c61c37 avec bono
5  8211339 avec mick
6  e137e9b premier commit
7  git checkout 8211339
8  cat rock.txt
9  bruce,mick
10 git checkout master
11 cat rock.txt
12 bruce,mick,bono
```

- Ligne 3 : `git log` fournit les commits avec leur id.

Figure: Etat du dépôt local avant checkout ligne 7

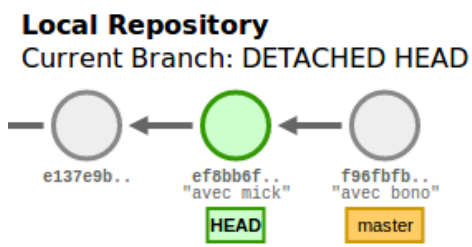


Back to the future (”Nom de Zeus”)

```
1  cat rock.txt
2  bruce,mick,bono
3  git log --oneline
4  8c61c37 avec bono
5  8211339 avec mick
6  e137e9b premier commit
7  git checkout 8211339
8  cat rock.txt
9  bruce,mick
10 git checkout master
11 cat rock.txt
12 bruce,mick,bono
```

- Ligne 3 : `git log` fournit les commits avec leur id.
- Ligne 7 : saut dans le passé, l’espace de travail contient l’état du second commit

Figure: Etat du dépôt local après checkout ligne 7

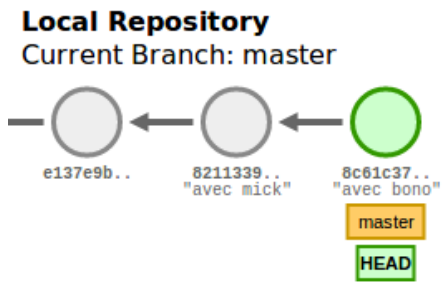


Back to the future (”Nom de Zeus”)

```
1  cat rock.txt
2  bruce,mick,bono
3  git log --oneline
4  8c61c37 avec bono
5  8211339 avec mick
6  e137e9b premier commit
7  git checkout 8211339
8  cat rock.txt
9  bruce,mick
10 git checkout master
11 cat rock.txt
12 bruce,mick,bono
```

- Ligne 3 : `git log` fournit les commits avec leur id.
- Ligne 7 : saut dans le passé, l’espace de travail contient l’état du second commit
- Ligne 10 : retour vers le futur, on se retrouve dans l’état du dernier commit

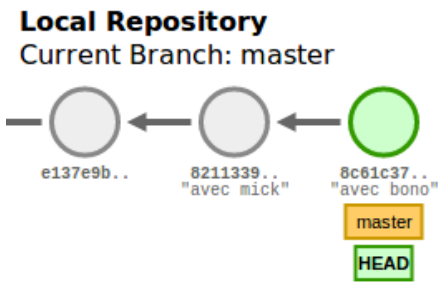
Figure: Etat du dépôt local après checkout master



Suppression de version

```
1  cat rock.txt
2  bruce,mick,bono
3  git reset HEAD^ --hard
4  cat rock.txt
5  bruce,mick
```

Figure: Etat du dépôt local au départ

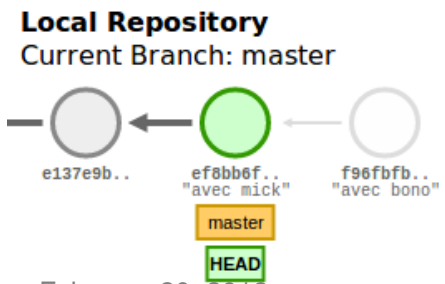


Suppression de version

- Ligne 3 : suppression du dernier commit, espace de travail modifié (option hard), les références `HEAD` et `MASTER` se déplacent.

```
1  cat rock.txt
2  bruce,mick,bono
3  git reset HEAD^ --hard
4  cat rock.txt
5  bruce,mick
```

Figure: Etat du dépôt local après reset ligne 3

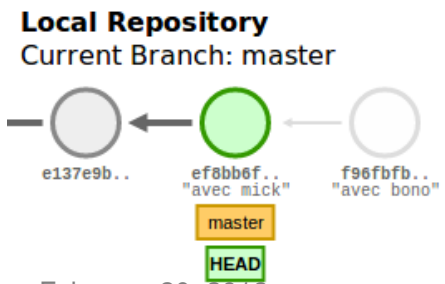


Suppression de version

```
1  cat rock.txt
2  bruce,mick,bono
3  git reset HEAD^ --hard
4  cat rock.txt
5  bruce,mick
```

- Ligne 3 : suppression du dernier commit, espace de travail modifié (option hard), les références `HEAD` et `MASTER` se déplacent.
- Sans l'option hard, l'espace de travail est inchangé et le fichier `rock.txt` est non suivi.

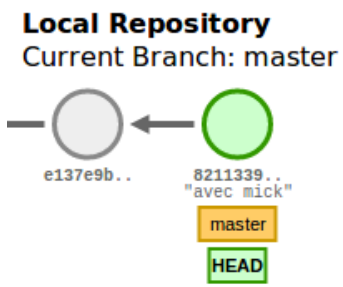
Figure: Etat du dépôt local après reset ligne 3



Branches et fusion

```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

Figure: État du dépôt local au départ

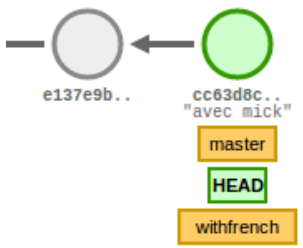


Branches et fusion

```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

- Ligne 1 : création d'une branche.

Figure: État du dépôt local après création branche

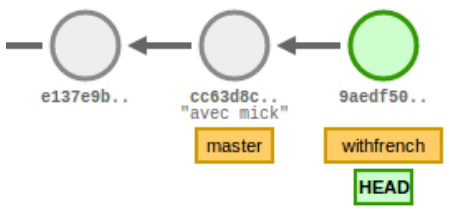


Branches et fusion

```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

- Ligne 1 : création d'une branche.
- Lignes 2-5 : travail dans cette branche

Figure: État du dépôt local après commit dans branche

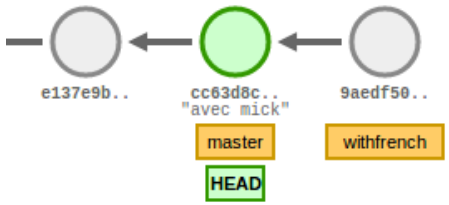


Branches et fusion

```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

- Ligne 1 : création d'une branche.
- Lignes 2-5 : travail dans cette branche
- Ligne 6 : retour branche principale

Figure: État du dépôt local après checkout master

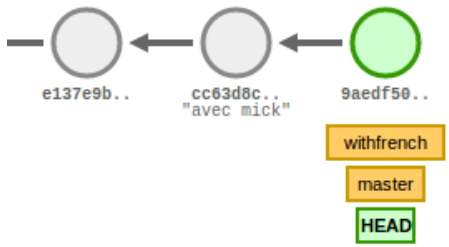


Branches et fusion

```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

- Ligne 1 : création d'une branche.
- Lignes 2-5 : travail dans cette branche
- Ligne 6 : retour branche principale
- Ligne 9 : fusion de `withfrench` dans `master`

Figure: État du dépôt local après merge

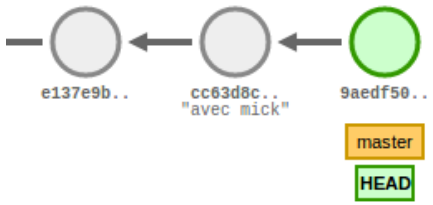


Branches et fusion

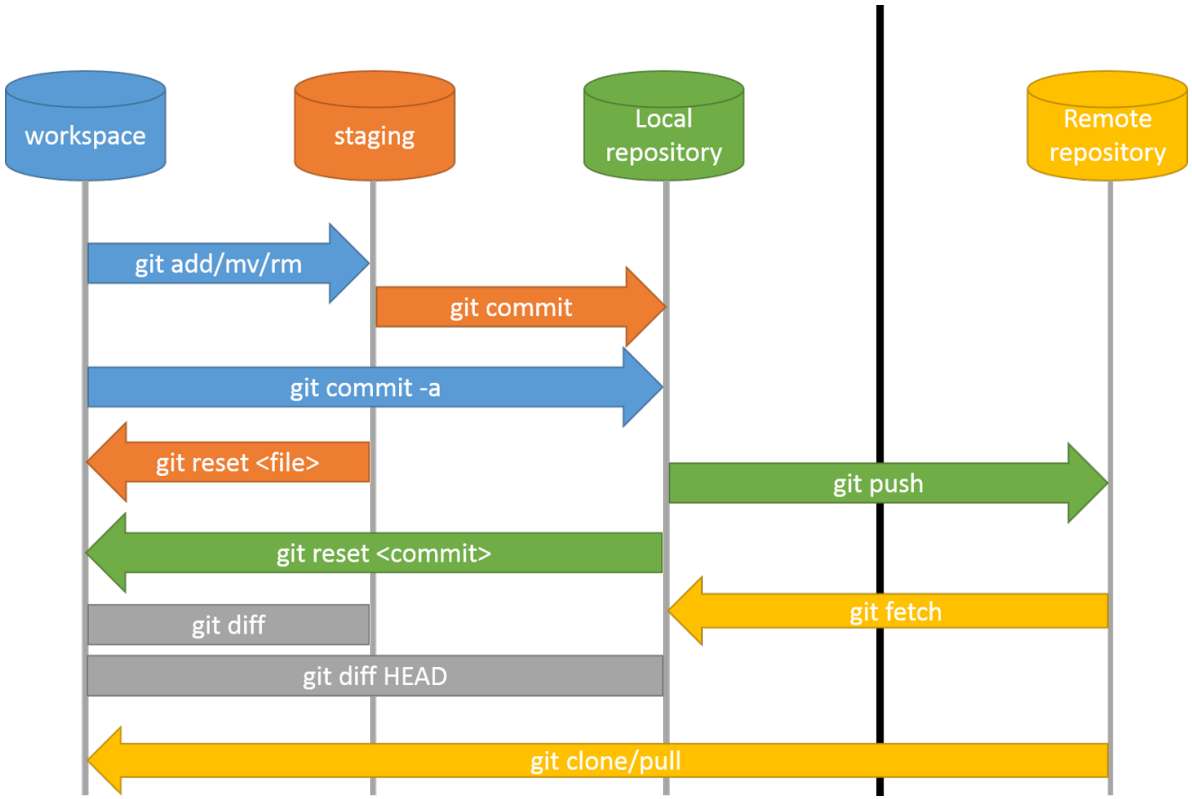
```
1  git branch withfrench
2  git checkout withfrench
3  echo
   ,johnny >> rock.txt
4  git add rock.txt
5  git commit -m "avec j."
6  git checkout master
7  cat rock.txt
8  bruce,mick
9  git merge withfrench
10 cat rock.txt
11 bruce,mick,johnny
12 git branch -d withfrench
```

- Ligne 1 : création d'une branche.
- Lignes 2-5 : travail dans cette branche
- Ligne 6 : retour branche principale
- Ligne 9 : fusion de `withfrench` dans `master`
- Ligne 12 : suppression de branche

Figure: État du dépôt local après suppression branche



Dépôt distant : clone, push, pull, fetch



Source : `https://github.com`

Initialisation du dépôt distant

1. Sur un serveur Git, se connecter et créer un projet vide.
   Exemples de serveur Git :
   - `https://archives.plil.fr`
   - `https://github.com`
2. Spécifiez les droits : qui peut accéder au projet,qui peut modifier,...
3. Récupérer l'adresse associée du projet.
   Ex:
   `https://archives.plil.fr/ocaron/demoCours.git`
4. Première alimentation dans le dépôt distant :

```
1  git remote add origin https://archives.plil.fr/ocaron/demoCours.git
2  git push -u origin master
```

Récupération et utilisation d'un dépôt distant

- Les utilisateurs autorisés peuvent récupérer le projet via la commande `clone` :

```
git clone https://archives.plil.fr/ocaron/demoCours.git
```

- Par la suite, on dispose des commandes suivantes :

git push	propage le dépôt local vers le dépôt distant.
git pull	opération duale, met à jour l'espace de travail
git push origin <nomBranche>	permet de propager une branche dans le dépôt distant

- Utilisation collaborative : faites un "pull" avant le "push".

Résolution de conflits

- Plusieurs développeurs peuvent travailler sur des sources partagés.

Résolution de conflits

- Plusieurs développeurs peuvent travailler sur des sources partagés.
- Cette activité collaborative peut engendrer des conflits lors de différentes activités (merge, commit, . . .).

Résolution de conflits

- Plusieurs développeurs peuvent travailler sur des sources partagés.
- Cette activité collaborative peut engendrer des conflits lors de différentes activités (merge, commit, . . .).
- L'outil GIT alerte sur ces conflits et génère dans les fichiers sources concernés des annotations précisant les différences et origines des modifications.

Résolution de conflits

- Plusieurs développeurs peuvent travailler sur des sources partagés.
- Cette activité collaborative peut engendrer des conflits lors de différentes activités (merge, commit, . . .).
- L'outil GIT alerte sur ces conflits et génère dans les fichiers sources concernés des annotations précisant les différences et origines des modifications.
- La résolution est manuelle et nécessite un dernier commit.

Résolution de conflits : illustration (1/2)

- Deux activités en parallèle :

```
1  git branch b1
2  git checkout b1
3  echo eddy >> rock.txt
4  git add rock.txt
5  git commit -m "b1 : avec eddy"
6  cat rock.txt
7  bruce,mick,johnny
8  eddy
```

```
1  git branch b2
2  git checkout b2
3  echo prince >> rock.txt
4  git add rock.txt
5  git commit -m "b2 : avec prince"
6  cat rock.txt
7  bruce,mick,johnny
8  prince
```

Résolution de conflits : illustration (2/2)

- Intégration des deux activités :

```
1   git checkout master
2   git merge b1
3   git merge b2
4   cat rock.txt
5   bruce,mick,johnny
6   <<<<<<< HEAD
7   eddy
8   =======
9   prince
10  >>>>>>> b2
```

- Ligne 2 : fusion dans master de branche b1, pas de conflit, ajout de ligne ”eddy”

Résolution de conflits : illustration (2/2)

- Intégration des deux activités :

```
1   git checkout master
2   git merge b1
3   git merge b2
4   cat rock.txt
5   bruce,mick,johnny
6   <<<<<<< HEAD
7   eddy
8   =======
9   prince
10  >>>>>>> b2
```

- Ligne 2 : fusion dans master de branche b1, pas de conflit, ajout de ligne ”eddy”
- Ligne 3 : la fusion de b2 provoque un conflit, le fichier `rock.txt` précise le conflit.

Résolution de conflits : illustration (2/2)

- Intégration des deux activités :

```
1  git checkout master
2  git merge b1
3  git merge b2
4  cat rock.txt
5  bruce,mick,johnny
6  <<<<<<< HEAD
7  eddy
8  =======
9  prince
10 >>>>>>> b2
```

- Ligne 2 : fusion dans master de branche b1, pas de conflit, ajout de ligne ”eddy”
- Ligne 3 : la fusion de b2 provoque un conflit, le fichier `rock.txt` précise le conflit.
- Il faut éditer le fichier (résolution du conflit,supprimer les annotations) puis faire add et commit

Conclusion

- Gestionnaire de versions, élément incontournable du développeur ou équipe de développeurs

Conclusion

- Gestionnaire de versions, élément incontournable du développeur ou équipe de développeurs
- GIT, outil performant et massivement utilisé

Conclusion

- Gestionnaire de versions, élément incontournable du développeur ou équipe de développeurs
- GIT, outil performant et massivement utilisé
- Ce cours est une **introduction** de Git

Conclusion

- Gestionnaire de versions, élément incontournable du développeur ou équipe de développeurs
- GIT, outil performant et massivement utilisé
- Ce cours est une **introduction** de Git
- GIT: outil texte mais de nombreuses extensions
  Ex: EGit pour environnement Eclipse