

Introduction au gestionnaire de versions GIT

Olivier Caron

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

<http://ocaron.polytech-lille.net>
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)

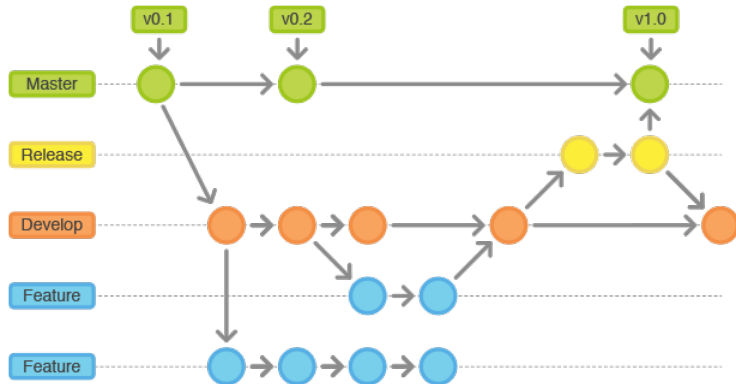
- Un **projet** de développement logiciel est une activité longue et complexe.
- Concerne plusieurs **fichiers**
- De multiples **itérations** sont nécessaires.
- A certains moments, on peut identifier des **versions** et/ou **variantes** du logiciel.
- Les erreurs sont possibles, **revenir en arrière**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

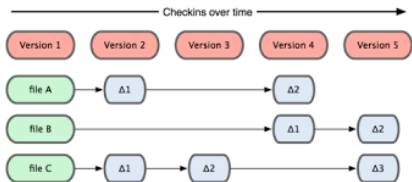
- Créé en 2005 par Linus Torvalds (également créateur de linux) pour gérer le noyau linux.
- Complètement distribué, rapide
- Fonctionne sans réseau **ET** avec réseau !

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.

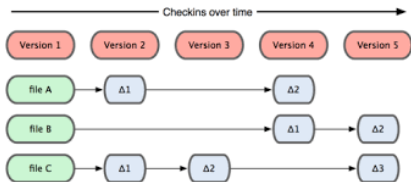
1ère originalité de GIT : des snapshots, pas des différences

- Autres systèmes de gestion de versions :

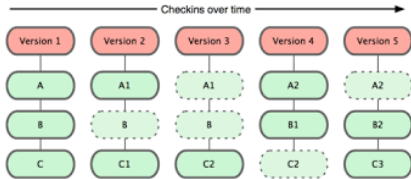


1ère originalité de GIT : des snapshots, pas des différences

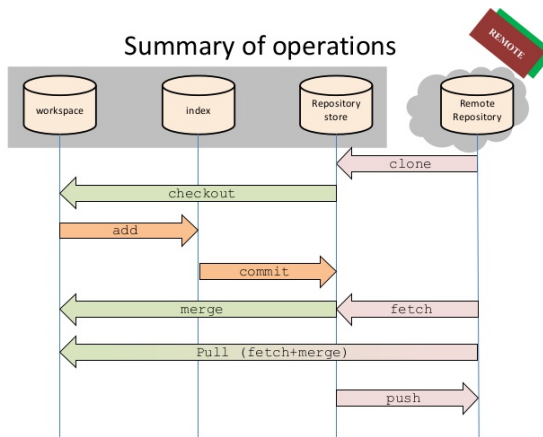
- Autres systèmes de gestion de versions :



- 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
```


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

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, ...

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, ...
- Disposer 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"

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

```
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
- Ligne 6 : ajout des fichiers dans l'espace `index`

FIGURE : État du dépôt local :

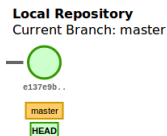
dépôt vide

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
- Ligne 7 : première version dans dépôt local
→ `commit_id`

FIGURE : État du dépôt local après commit :



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 (et leur `id`)

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 (et leur `id`)
 - `git show <commit.id>` décrit le commit passé en paramètre

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

Local Repository
Current Branch: master

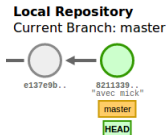


Continuons à développer

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

```
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 après commit ligne 3

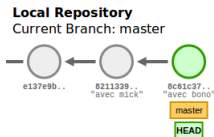


Continuons à développer

```
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 4-6 : évolution du fichier, troisième commit

FIGURE : État du dépôt local après commit ligne 6



Back to the future ("Nom de Zeus")

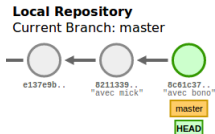
```

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")

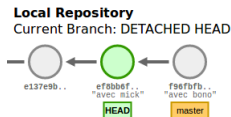
```

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")

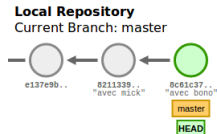
```

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

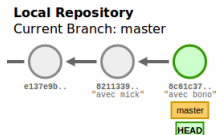
FIGURE : Etat du dépôt local après checkout master



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

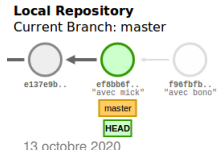


Suppression de version

- 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

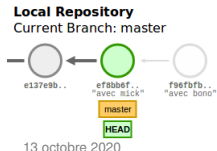


Suppression de version

```
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.

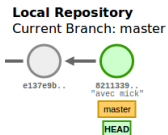
FIGURE : Etat du dépôt local après reset ligne 3



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_lj."
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

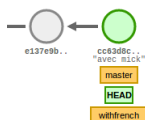


Branches et fusion

```
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

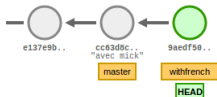
```

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



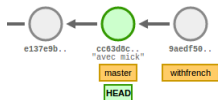
Branches et fusion

```

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

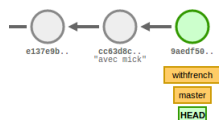
```

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

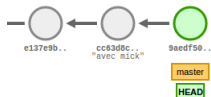
```

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

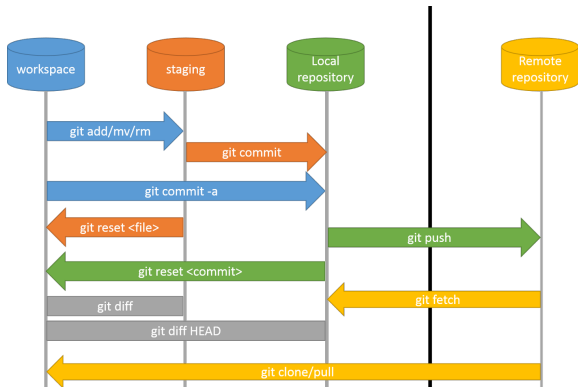
```

- 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 :

Initialisation du dépôt distant

- 1 Sur un serveur Git, se connecter et créer un projet vide.
Exemples de serveur Git :
 - `https://gitlab.univ-lille.fr`

Initialisation du dépôt distant

- 1 Sur un serveur Git, se connecter et créer un projet vide.

Exemples de serveur Git :

- `https://gitlab.univ-lille.fr`
- `https://github.com`

Initialisation du dépôt distant

- 1 Sur un serveur Git, se connecter et créer un projet vide.
Exemples de serveur Git :
 - `https://gitlab.univ-lille.fr`
 - `https://github.com`
- 2 Spécifiez les droits : qui peut accéder au projet, qui peut modifier,...

Initialisation du dépôt distant

- 1 Sur un serveur Git, se connecter et créer un projet vide.

Exemples de serveur Git :

- `https://gitlab.univ-lille.fr`
- `https://github.com`

- 2 Spécifiez les droits : qui peut accéder au projet, qui peut modifier,...

- 3 Récupérer l'adresse associée du projet. Exemple :

`https://gitlab.univ-lille/olivier.caron/democours.git`

Initialisation du dépôt distant

- 1 Sur un serveur Git, se connecter et créer un projet vide.

Exemples de serveur Git :

- `https://gitlab.univ-lille.fr`
- `https://github.com`

- 2 Spécifiez les droits : qui peut accéder au projet, qui peut modifier,...

- 3 Récupérer l'adresse associée du projet. Exemple :

```
https://gitlab.univ-lille/olivier.caron/democours.git
```

- 4 Première alimentation dans le dépôt distant :

```
git remote add origin https://gitlab.univ-lille.fr/olivier.caron/democours.git  
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://gitlab.univ-lille.fr/olivier.caron/democours.git
```

Récupération et utilisation d'un dépôt distant

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

```
git clone https://gitlab.univ-lille.fr/olivier.caron/democours.git
```

- Par la suite, on dispose des commandes suivantes :

Récupération et utilisation d'un dépôt distant

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

```
git clone https://gitlab.univ-lille.fr/olivier.caron/democours.git
```

- Par la suite, on dispose des commandes suivantes :

git push propage le dépôt local vers le dépôt distant.

Récupération et utilisation d'un dépôt distant

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

```
git clone https://gitlab.univ-lille.fr/olivier.caron/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

Récupération et utilisation d'un dépôt distant

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

```
git clone https://gitlab.univ-lille.fr/olivier.caron/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

Récupération et utilisation d'un dépôt distant

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

```
git clone https://gitlab.univ-lille.fr/olivier.caron/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 (1/2)

- 1 seul développeur mais deux développements en parallèle :

Résolution de conflits : illustration 1 (1/2)

- 1 seul développeur mais deux développements en parallèle :

Résolution de conflits : illustration 1 (1/2)

- 1 seul développeur mais deux développements 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
```

Résolution de conflits : illustration 1 (1/2)

- 1 seul développeur mais deux développements en parallèle :

1	git branch b1	1	git branch b2
2	git checkout b1	2	git checkout b2
3	echo eddy >> rock.txt	3	echo prince >> rock.txt
4	git add rock.txt	4	git add rock.txt
5	git commit -m "b1 : avec eddy"	5	git commit -m "b2 : avec prince"
6	cat rock.txt	6	cat rock.txt
7	bruce , mick , johnny	7	bruce , mick , johnny
8	eddy	8	prince

Résolution de conflits : illustration 1 (2/2)

- Intégration des deux activités :

- Ligne 2 : fusion de master dans b1, ici : aucune activité dans master depuis création branche b1.

```
1 git checkout b1
2 git merge master
3 → déjà à jour
4 git checkout master
5 git merge b1
6 git checkout b2
7 git merge master
8 cat rock.txt
9 bruce , mick , johnny
10 <<<<<<<< HEAD
11 eddy
12 =====
13 prince
14 >>>>>>> b2
```

Résolution de conflits : illustration 1 (2/2)

- Intégration des deux activités :

```
1 git checkout b1
2 git merge master
3 → déjà à jour
4 git checkout master
5 git merge b1
6 git checkout b2
7 git merge master
8 cat rock.txt
9 bruce , mick , johnny
10 <<<<<<<< HEAD
11 eddy
12 =====
13 prince
14 >>>>>>> b2
```

- Ligne 2 : fusion de master dans b1, ici : aucune activité dans master depuis création branche b1.
- Ligne 5 : fusion de branche b1 dans master : pas de conflit donc ajout de ligne "eddy"

Résolution de conflits : illustration 1 (2/2)

- Intégration des deux activités :

```

1 git checkout b1
2 git merge master
3 → déjà à jour
4 git checkout master
5 git merge b1
6 git checkout b2
7 git merge master
8 cat rock.txt
9 bruce , mick , johnny
10 <<<<<<<< HEAD
11 eddy
12 =====
13 prince
14 >>>>>>> b2
  
```

- Ligne 2 : fusion de master dans b1, ici : aucune activité dans master depuis création branche b1.
- Ligne 5 : fusion de branche b1 dans master : pas de conflit donc ajout de ligne "eddy"
- Ligne 7 : la fusion de master dans b2 provoque un conflit, le fichier `rock.txt` précise le conflit. Il faut alors éditer le fichier (résolution du conflit, supprimer les annotations) commiter puis fusionner dans master :

```

gedit rock.txt — gestion des conflits
git add rock.txt
git commit -m "solution_à_conflits"
git checkout master
git merge b2
  
```

Résolution de conflits : illustration 2 (1/3)

- Développement collaboratif, 2 utilisateurs
- Au départ, les 2 utilisateurs ont le même état
- Ce que fait l'utilisateur `user1` :

```
1 git checkout master
2 cat rock.txt
3 bruce , mick , johnny
4 eddy
5 prince
6 echo "muse" >> rock.txt
7 git add rock.txt
8 git commit -m "user1:muse"
9 git pull
10 git push
```

- Lignes 1-8 : travail sur branche principale (master), commit dans dépôt local

Résolution de conflits : illustration 2 (1/3)

- Développement collaboratif, 2 utilisateurs
- Au départ, les 2 utilisateurs ont le même état
- Ce que fait l'utilisateur `user1` :

```
1 git checkout master
2 cat rock.txt
3 bruce , mick , johnny
4 eddy
5 prince
6 echo "muse" >> rock.txt
7 git add rock.txt
8 git commit -m "user1:muse"
9 git pull
10 git push
```

- Lignes 1-8 : travail sur branche principale (master), commit dans dépôt local
- Ligne 9 : vérification de conflits avec dépôt distant : ici, pas de conflit.

Résolution de conflits : illustration 2 (1/3)

- Développement collaboratif, 2 utilisateurs
- Au départ, les 2 utilisateurs ont le même état
- Ce que fait l'utilisateur `user1` :

```
1 git checkout master
2 cat rock.txt
3 bruce , mick , johnny
4 eddy
5 prince
6 echo "muse" >> rock.txt
7 git add rock.txt
8 git commit -m "user1:muse"
9 git pull
10 git push
```

- Lignes 1-8 : travail sur branche principale (master), commit dans dépôt local
- Ligne 9 : vérification de conflits avec dépôt distant : ici, pas de conflit.
- Ligne 10 : propagation dépôt local vers dépôt distant

Résolution de conflits : illustration 2 (2/3)

- Ce que fait l'utilisateur `user2`, un peu plus tard :

```
1 cat rock.txt
2 bruce , mick , johnny
3 eddy
4 prince
5 echo "u2" >> rock.txt
6 git add rock.txt
7 git commit -m "user2:u2"
8 git pull
9 gedit rock.txt
10 git add rock.txt
11 git commit -m "conflitResolu"
12 git push
```

- Lignes 1-7 : travail sur branche principale (master), commit dans dépôt local

Résolution de conflits : illustration 2 (2/3)

- Ce que fait l'utilisateur `user2`, un peu plus tard :

```
1 cat rock.txt
2 bruce , mick , johnny
3 eddy
4 prince
5 echo "u2" >> rock.txt
6 git add rock.txt
7 git commit -m "user2:u2"
8 git pull
9 gedit rock.txt
10 git add rock.txt
11 git commit -m "conflitResolu"
12 git push
```

- Lignes 1-7 : travail sur branche principale (master), commit dans dépôt local
- Ligne 8 : vérification de conflits avec dépôt distant : ici, **conflit détecté**.

Résolution de conflits : illustration 2 (2/3)

- Ce que fait l'utilisateur `user2`, un peu plus tard :

```
1 cat rock.txt
2 bruce , mick , johnny
3 eddy
4 prince
5 echo "u2" >> rock.txt
6 git add rock.txt
7 git commit -m "user2:u2"
8 git pull
9 gedit rock.txt
10 git add rock.txt
11 git commit -m "conflitResolu"
12 git push
```

- Lignes 1-7 : travail sur branche principale (master), commit dans dépôt local
- Ligne 8 : vérification de conflits avec dépôt distant : ici, **conflit détecté**.
- Lignes 9-11 : résolution conflit, dépôt local modifié

Résolution de conflits : illustration 2 (2/3)

- Ce que fait l'utilisateur `user2`, un peu plus tard :

```
1 cat rock.txt
2 bruce , mick , johnny
3 eddy
4 prince
5 echo "u2" >> rock.txt
6 git add rock.txt
7 git commit -m "user2:u2"
8 git pull
9 gedit rock.txt
10 git add rock.txt
11 git commit -m "conflitResolu"
12 git push
```

- Lignes 1-7 : travail sur branche principale (master), commit dans dépôt local
- Ligne 8 : vérification de conflits avec dépôt distant : ici, **conflit détecté**.
- Lignes 9-11 : résolution conflit, dépôt local modifié
- Ligne 12 : propagation dépôt local vers dépôt distant

Résolution de conflits : illustration 2 (3/3)

- Pour revenir à un état identique entre les deux développeurs, l'utilisateur `user1` doit faire un `git pull` (ligne 6) :

```
1  cat rock.txt
2  bruce ,mick ,johnny
3  eddy
4  prince
5  muse
6  git pull
7  cat rock.txt
8  bruce ,mick ,johnny
9  eddy
10 prince
11 u2:ok muse:beurk — ici , la résolution du conflit de user2
12 qui n'engage que lui
```

Développement collaboratif

Règle n°1

Faire un git pull avant un git push

Développement collaboratif

Règle n°1

Faire un `git pull` avant un `git push`

Règle n°2

Au début d'une séance de projet, faire un `git pull` pour être certain de travailler avec la dernière version du projet

Développement collaboratif

Règle n°1

Faire un `git pull` avant un `git push`

Règle n°2

Au début d'une séance de projet, faire un `git pull` pour être certain de travailler avec la dernière version du projet

Règle n°3

Ne jamais oublier les deux premières règles

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