

git pour les novices

<?="Atilla" :?>

Pierre Sudron

EISTI

20 septembre 2012

git est un système de gestion de sources

- gérer l'évolution d'un code
- partager efficacement son travail en équipe
- garder un historique de l'évolution d'un projet
- travailler en parallèle



Autres systèmes existants

- Subversion (svn)
- Mercurial (Hg)
- Bazaar (bzt)
- autres systèmes propriétaires



Plus particulièrement sur git

- créé pour le noyau Linux
- développé par Junio Hamano
- distribué et très flexible
- adoption rapide et massive depuis 2005

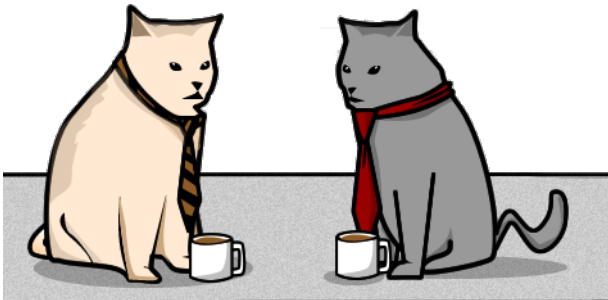


Qu'est-ce que git peut m'apporter ?

- suivi précis de l'avancement d'un projet
- souplesse dans l'évolution
- facilité de mise en commun



Intégration douloureuse : un projet qui fait plouf



Bob et Bob travaillent sur un projet
et se répartissent les tâches

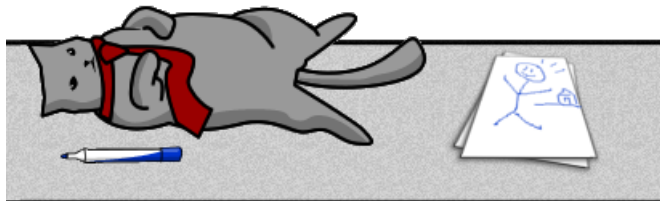


À deux jours du livrable...



Bob a bien avancé sur sa partie





Bob pas tellement
mais il a fait quelque chose au moins

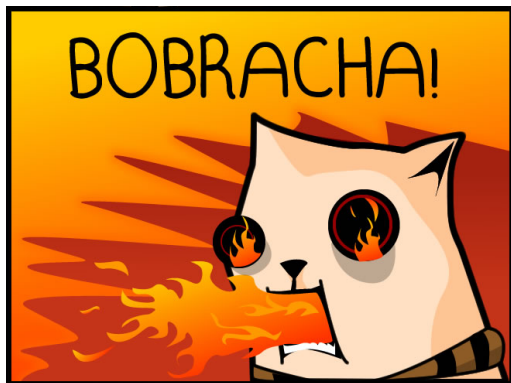


C'est l'heure de mettre en commun !



Et quelle surprise, ça marche pas !



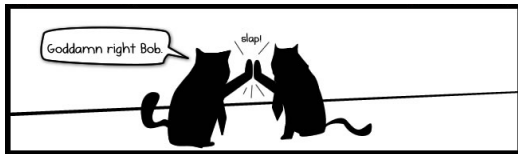


Et à 24 heures du rendu, c'est un peu cuit...



L'intégration progressive

- git incite à mettre en commun très régulièrement
- en cas d'ennui, il est possible de revenir à la dernière version fonctionnelle
- de là, il est facile de voir les modifications ultérieures et isoler le code en cause



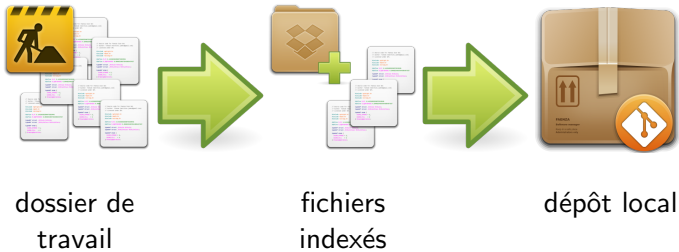
De quoi ai-je besoin ?

Ouvrir un terminal
(mais sous Linux, hein...)



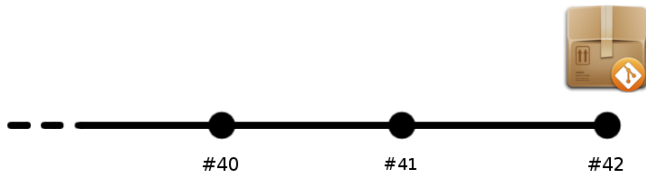
Principes de fonctionnement

En **local**, on travaille avec 3 éléments :



Contenu d'un dépôt git

Succession de **commits**



Un commit est un jeu de modifications, c'est l'*atome* de git.



Mise en commun

Synchronisation entre le dépôt local et un dépôt distant



Partie 1 : C'est un voyage qu'il faut commencer seul

versionnage d'un simple fichier texte

```
// Source code for Faenza Icon Set
// Author: Tihuan (matthieu.james@gmail.com)
// licenced under GPL

#include <gtk/gtk.h>
#include <math.h>
#include <string.h>

#define M_PI 3.14159265358979323846
#define M_SQRT2 M_SQRT2

typedef struct _HcStyle HcStyle;
typedef struct _HcStyleClass HcStyleClass;

typedef enum {
    CASING_SYMBOL = 1,
    CASING_FILL = 2
} DrawingOperation;
```



Configuration de git : qui suis-je ?

git retrace l'évolution du projet ainsi :

qui a fait **quoi**, et **quand**

Pour définir **qui** est l'utilisateur :

```
git config --global user.name "Votre nom"  
git config --global user.email "root@atilla.org"
```

Pour activer la couleur dans le terminal :

```
git config --global color.ui auto
```



Préparer le projet

Se placer dans le dossier de travail (le créer si besoin) :

```
mkdir ~/formation_git  
cd ~/formation_git/
```

Initialiser git dans ce répertoire :

```
git init
```



Comment ça se présente



répertoire de travail : vide



fichiers indexés : vide



dépôt local : initialisé et vide



Créer un fichier

Sans changer de dossier, créer un fichier **README** avec le contenu suivant :

```
Formation git
```

Enregistrer le fichier



Comment ça se présente



répertoire de travail : README



fichiers indexés : vide



dépôt local : initialisé et vide



Constater les changements avec git status

```
git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
nothing added to commit but untracked files present (use "git add" to track)
```

README n'est pas indexé, on nous propose d'utiliser git add



Indexer un fichier avec git add

```
git add README
```

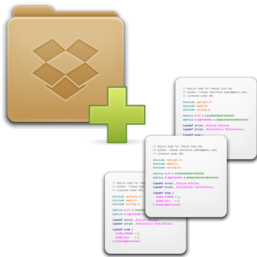
```
// Source code for Fabrice Ison test
// Author: Tihomir Matkovic (matkovic@gmail.com)
// Licensed under GPL

#include <git/git.h>
#include <math.h>
#include <string.h>

void main (int argc, char **argv)
{
    struct _Mobyli Mobyli;
    struct _MobyliClass MobyliClass;

    typedef struct {
        int x;
        int y;
    } Mobyli;

    MobyliClass.x = 1;
    MobyliClass.y = 2;
    MobyliClass.z = 3;
}
```



Comment ça se présente



répertoire de travail : README



fichiers indexés : README (nouveau fichier)



dépôt local : initialisé et vide



Constater les changements avec git status

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#
```

README est dans l'index en tant que nouveau fichier



Notre premier git commit

On va mettre à jour le dépôt local et laisser une trace de la création du fichier README

```
git commit -m "Ajout du fichier README"
```

Tout commit est décrit par un **message**, il doit être :

- **précis**
- **concis**



Comment ça se présente



répertoire de travail : README



fichiers indexés : vide



dépôt local : commit #1

(vérifier avec git status)



Historique des commits



Commit #1 :
Ajout du fichier README



Poursuivons le travail...

Ajouter une ligne à la fin du fichier **README** :

```
Formation git  
Avec l'association Atilla
```

Enregistrer le fichier



Comment ça se présente



répertoire de travail : README (modifié)



fichiers indexés : vide



dépôt local : commit #1



Constater les changements avec git status

```
git status
```

Les fichiers modifiés sont détectés ; on propose de les indexer avec git add

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README
#
no changes added to commit (use "git add" and/or "git commit -a")
```



Indexation du fichier modifié

```
git add README
```



répertoire de travail : README (modifié)



fichiers indexés : **README (modifié)**



dépôt local : commit #1

(vérifier avec git status)



Validation des changements

```
git commit -m "Modification du fichier README"
```



répertoire de travail : README (modifié)



fichiers indexés : vide



dépôt local : commit #1, commit #2

(vérifier avec git status)



Historique des commits



Historique des commits

```
git log
```

```
commit 82aad05924900b273d50f3b55e7d905896931e8d
```

```
Author: Pierre Sudron <sudronpier@eisti.eu>
```

```
Date: Thu Aug 23 20:16:30 2012 +0200
```

```
Modification du fichier README
```

```
commit b1098965de8d0948104ceb657be01fb9a381860a
```

```
Author: Pierre Sudron <sudronpier@eisti.eu>
```

```
Date: Thu Aug 23 18:36:22 2012 +0200
```

```
Ajout du fichier README
```



Export vers un dépôt git en ligne

Nous utiliserons une instance **GitLab** disponible à l'adresse :

`http://gitlab.etude.eisti.fr/`

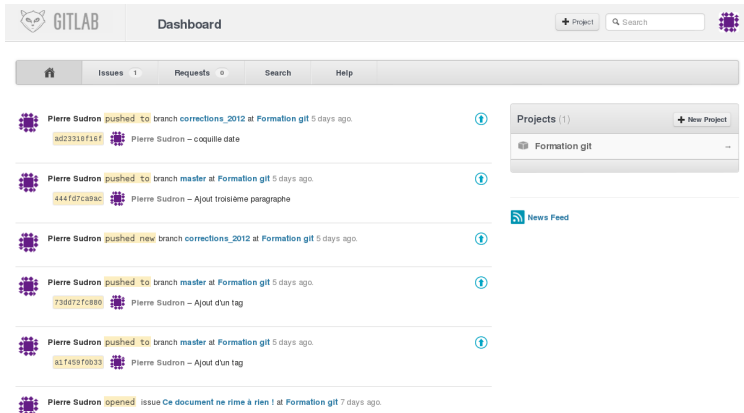


Il existe de nombreux sites permettant d'héberger vos projets git, dont :

- GitHub
- Gitorious



Se connecter et ajuster son profil sur GitLab



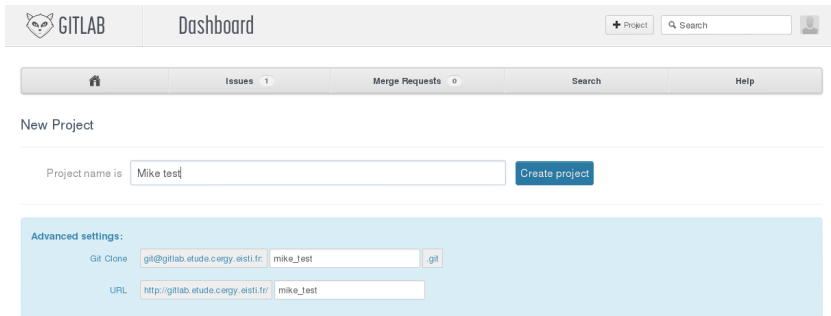
The screenshot shows the GitLab Dashboard for a user named Pierre Sudron. The top navigation bar includes the GitLab logo, the word "Dashboard", a "+ Project" button, a search bar, and a user profile icon. Below this is a secondary navigation bar with tabs for Home, Issues (1), Requests (0), Search, and Help. The main content area displays a list of recent activity:

- Pierre Sudron pushed to branch corrections_2012 at Formation git 5 days ago.** (Info icon) Commit: ad23318f16f. Description: Pierre Sudron – coquille date.
- Pierre Sudron pushed to branch master at Formation git 5 days ago.** (Info icon) Commit: 444fd7ca9ac. Description: Pierre Sudron – Ajout troisième paragraphe.
- Pierre Sudron pushed new branch corrections_2012 at Formation git 5 days ago.** (Info icon)
- Pierre Sudron pushed to branch master at Formation git 5 days ago.** (Info icon) Commit: 73dd72fc880. Description: Pierre Sudron – Ajout d'un tag.
- Pierre Sudron pushed to branch master at Formation git 5 days ago.** (Info icon) Commit: a11459f0b33. Description: Pierre Sudron – Ajout d'un tag.
- Pierre Sudron opened issue Ce document ne rime à rien ! at Formation git 7 days ago.**


On the right side, there is a "Projects (1)" section with a "+ New Project" button and a list containing "Formation git". Below that is a "News Feed" section with an RSS icon.



Créer un projet sur GitLab



The screenshot shows the GitLab Dashboard interface. At the top, there is a navigation bar with the GitLab logo, the word 'Dashboard', a '+ Project' button, a search bar, and a user profile icon. Below this is a secondary navigation bar with links for Home, Issues (1), Merge Requests (0), Search, and Help. The main content area is titled 'New Project' and contains a form. The form has a label 'Project name is' followed by a text input field containing 'Mike test' and a blue 'Create project' button. Below the form is a light blue section titled 'Advanced settings:' which contains two rows of input fields. The first row is for 'Git Clone' with a dropdown menu set to 'git@gitlab.etude.cergy.eisti.fr:', a text input field containing 'mike_test', and a dropdown menu set to '.git'. The second row is for 'URL' with a dropdown menu set to 'http://gitlab.etude.cergy.eisti.fr/' and a text input field containing 'mike_test'.

 **GITLAB** Dashboard + Project

[Home](#) [Issues 1](#) [Merge Requests 0](#) [Search](#) [Help](#)

New Project

Project name is Create project

Advanced settings:

Git Clone

URL



Créer une clé SSH

Ouvrez un nouveau terminal :

```
ssh-keygen -t rsa -C "email@troll.com"
```

La clé SSH permet :

- de vous identifier formellement
- de crypter de transfert de code entre vous et le serveur



Indiquer la clé publique à GitLab

Le RSA est un cryptage asymétrique, et crée un jeu de deux clés

- une clé privée
- une **clé publique**

Pour afficher la clé publique :

```
cat .ssh/id_rsa.pub
```

- ajouter la clé publique au trousseau GitLab

(vous pouvez fermer le second terminal et revenir au précédent)

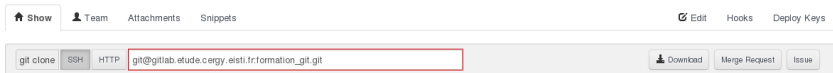


Ajouter le dépôt dans votre projet

```
git remote add origin [adresse du depot]
```

- **origin** : nom donné au serveur distant par convention

Vous trouverez l'adresse de votre dépôt sur sa page d'accueil.



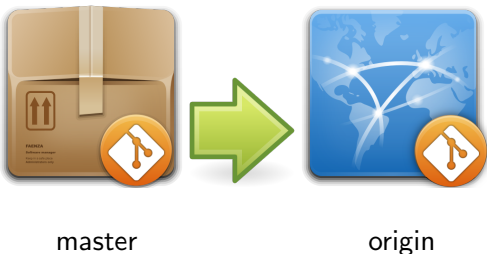
The screenshot shows the top navigation bar of a GitLab repository page. On the left, there are tabs for 'Show', 'Team', 'Attachments', and 'Snippets'. On the right, there are links for 'Edit', 'Hooks', and 'Deploy Keys'. Below the navigation bar is a toolbar with 'git clone' and 'SSH' buttons. A text input field contains the SSH URL: 'git@gitlab.etude.cergy.eisti.fr:formation_git.git'. To the right of the input field are buttons for 'Download', 'Merge Request', and 'Issue'.



Mettre le projet en ligne !

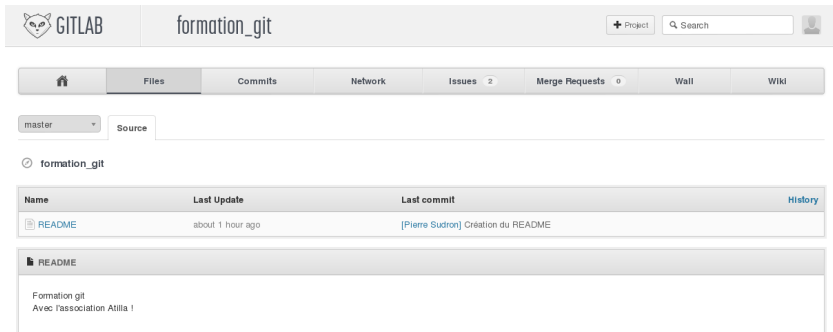
```
git push -u origin master
```

- **origin** : nom donné au serveur distant par convention
- **master** : nom donné au dépôt local par défaut





Voir le résultat sur GitLab

Il ne reste plus qu'à constater le résultat avec satisfaction !



The screenshot shows the GitLab interface for a project named 'formation_git'. At the top, there is a navigation bar with the GitLab logo, the project name, a '+ Project' button, and a search bar. Below this is a secondary navigation bar with tabs for 'Files', 'Commits', 'Network', 'Issues' (with a count of 2), 'Merge Requests' (with a count of 0), 'Wall', and 'Wiki'. The 'Files' tab is selected. Underneath, there is a dropdown menu for branches (currently showing 'master') and a 'Source' button. The main content area shows the project name 'formation_git' and a table of files. The table has columns for 'Name', 'Last Update', 'Last commit', and 'History'. One file is listed: 'README', updated 'about 1 hour ago' by '[Pierre Sudron]' with the commit message 'Création du README'. Below the table, there is a preview of the README file content, which reads: 'Formation git' and 'Avec l'association Attila !'.

| Name | Last Update | Last commit | History |
|--|------------------|--|---------|
|  README | about 1 hour ago | [Pierre Sudron] Création du README | |

 README

Formation git
Avec l'association Attila !



avant de continuer...

la Pause !



Partie 2 : Travailler à plusieurs

découverte des fonctionnalités de partage



De quoi ai-je besoin ?

logiciel Meld

prenez un moment pour l'installer avant de continuer...





Accès aux projets

Un projet n'est visible que pour les membres de l'équipe en charge.

🏠 Show **👤 Team** Attachments Snippets 🔗 Edit Hooks Deploy Keys

Team Members (2)
Read more about project permissions [here](#) New Team Member

| User | Permissions |
|---|-------------|
|  Pierre Sudron pierre.sudron@eisti.fr Project Owner | Master |
|  Thomas Humbert thomas.humbert@eisti.fr | Developer |

Les équipes comportent différents niveaux de responsabilité :

- Master
- Developer
- Reporter
- Guest



Cloner un dépôt existant

- récupérer l'adresse du dépôt



- cloner le dépôt (cette opération crée le répertoire du projet)

```
git clone [adresse]
```



Cloner un dépôt existant

Le projet **Participants formation** contient un seul fichier dans lequel chacun va ajouter son nom.

- assurez-vous que vous êtes inscrit au projet comme développeur
- clonez le dépôt sur votre machine
- tout le monde doit être à la même version



Allons y...

- ajoutez votre nom à la fin du fichier
- faites un commit
- essayer de push...

```
To git@gitlab.etude.cergy.eisti.fr:formation_git.git
! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'git@gitlab.etude.cergy.eisti.fr:formation_git.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

oui, ça veut dire
PA - TA - TRAS !



Qu'est-ce que j'ai fait pour mériter ça ?

Si il y a un mot à retenir dans le message d'erreur :

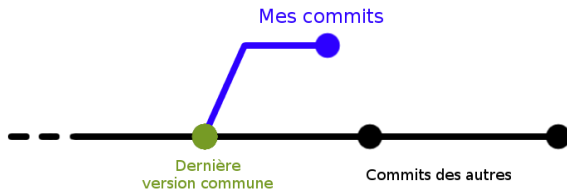
```
! [rejected]          master -> master (non-fast-forward)
```



fast-forward signifie que vous écraseriez les commits d'autres si git vous laissait push.



Qu'est-ce qui s'est passé ?



Un fichier a été modifié et mis en ligne depuis votre dernier pull.



Comment s'en sortir ?

On souhaiterait appliquer nos commits à la suite de ceux des autres.



C'est une opération de **rebase**.



Chacun son tour

- récupérer les derniers commits en ligne et rebaser nos commits à partir de là

```
git pull --rebase
```



- cette opération peut soulever un **conflit** bloquant car plusieurs personnes ont modifié le même fichier



Les conflits

Un **conflit** a lieu quand on cherche à mettre en commun deux fichiers dont les mêmes sections ont été modifiées par deux personnes.

| | | |
|---|-----|----------------------------------|
| Bonjour les gens. | | Bonjour les gens. |
| Il fait beau aujourd'hui. | → ← | Il fait moche aujourd'hui. |
| J'aime le sport. Je mange équilibré. | → ← | Je mange au McDo tous les jours. |



Résoudre un conflit bloquant un rebase

Le message d'erreur du "pull --rebase nous indique les deux étapes à suivre :

- corriger les conflits dans les fichiers continus
- achever et valider le rebase avec

```
git rebase --continue
```



Gérer les conflits avec une interface

Pour résoudre le conflit nous allons utiliser l'ONU **Meld**. Notez qu'il existe énormément de logiciels équivalents.

- définir Meld comme outil de gestion de conflit

```
git config --global merge.tool meld
```

- commencer à gérer le conflit

```
git mergetool
```



Gérer des conflits avec une interface

Meld (comme ses équivalents) présente sur trois colonnes, trois versions du fichier :

- **locale** à gauche : votre dernière version de la branche réceptrice
- **remote** à droite : la dernière version de la branche à fusionner
- **origin** au centre : la dernière version commune aux deux branches

Dans la pratique, on enregistrera le résultat voulu dans la **colonne centrale**, c'est à dire sur **origin**.



Finaliser la résolution de conflits

mergetool créé un fichier de sauvegarde <fichier>.orig, on peut les supprimer une fois la session mergetool terminée

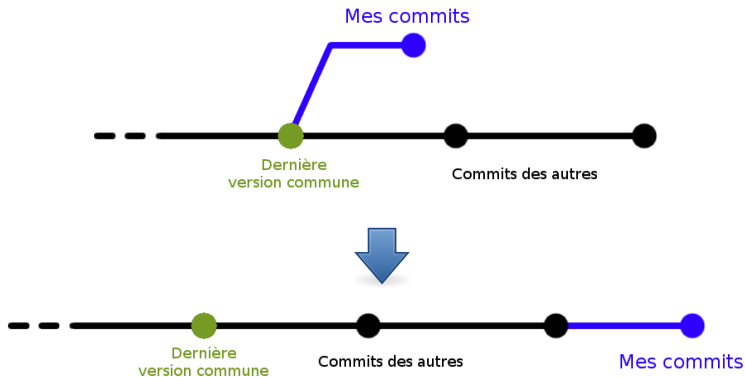
Il reste à terminer l'opération de rebase

```
git rebase --continue
```



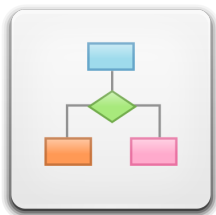
Partager son travail

Il n'y a plus de problème de **fast forward** : on peut désormais push.



Partie 3 : Comment ça va vieille branche ?

développer en parallèle



De quoi ai-je besoin ?

logiciel Meld



C'est l'histoire d'un prof...

Jean-Paul a fini de rédiger le poly du cours de cette année, en \LaTeX bien entendu !

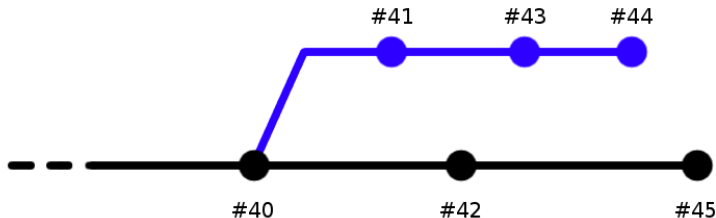
Il souhaite continuer à rajouter des parties pour l'année suivante tout en gardant une version de cette année afin de corriger les coquilles que ses élèves lui signalent.

Mais comme Jean Paul n'est pas très organisé, nous allons l'aider à mieux gérer son texte de cours avec **git** et les **branches** !



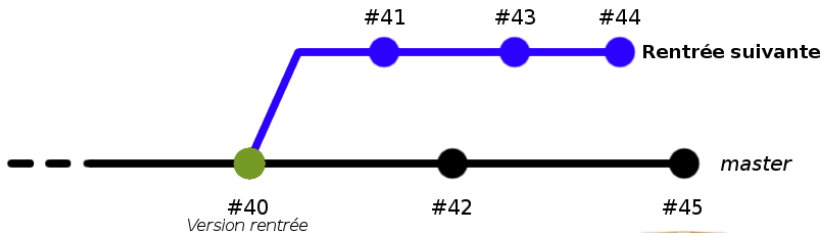
Le principe des branches

Fourche dans le processus de développement du projet. Une branche démarre à partir d'un commit donné.



Le principe des branches

Nous allons créer une branche dédiée aux corrections des fautes d'orthographe sur une branche partant de la version distribuée du poly.



Préparation du projet

- créer un dossier `formation_git_branches`
- déplacer le fichier `cours.tex` fourni dans ce dossier

```
cd formation_git_branches
```

```
git init
```

```
git add cours.tex
```

```
git commit -m "Version rentrée"
```



Créer une branche

La branche est créée à partir du dernier commit local (HEAD).

```
git branch rentrée_suivante
```

Pour lister les branches existantes :

```
git branch
```

```
* master  
  rentrée_suivante
```



Sauter de branche en branche

```
git checkout rentree_suivante
```

La commande git branch donne le résultat suivant :

```
master
* rentree_suivante
```



Ajouter un nouveau paragraphe pour la rentrée prochaine

Jean-Paul, studieux comme toujours, ne tarde pas à rajouter du contenu pour l'année prochaine.

Éditer le fichier `cours.tex` en ajoutant une ligne à la fin.
Sauvegarder, fermer l'éditeur de texte et commiter.

```
git add cours.tex  
git commit -m "Nouveau paragraphe"
```



Revenir sur la branche principale pour une correction

Mince! On a signalé la présence de fautes d'orthographe dans le poly de cette année.

Retourner sur la branche principale

```
git checkout master
```

Ouvrir le fichier `cours.tex` avec un éditeur. Vérifier qu'il s'agit bien de la version de cette rentrée.

Corriger les fautes, enregistrer, commiter.



Visualiser les commits selon leur branche

```
git log --oneline --graph --all
```

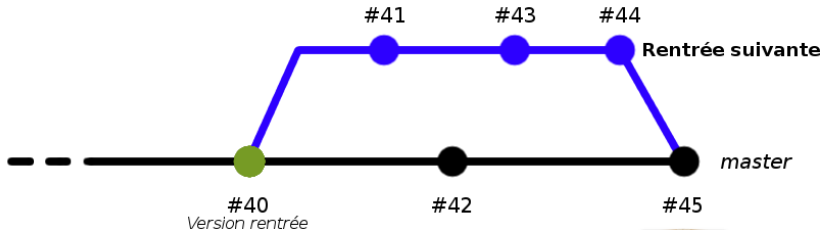
```
* 7fe9bbb Correction de fautes de frappe  
| * 45e6ac9 Nouveau paragraphe  
|/  
* 194a5b4 Version rentrée
```

le log se lit de bas en haut



Fusionner deux branches

La rentrée suivante est arrivée (ça passe vite), et Jean-Paul voudrait intégrer ses nouvelles parties tout en conservant les corrections faites en cours d'année.



Fusionner deux branches

Se placer dans la branche qui va "intégrer" les commits de l'autre

```
git checkout master
```

Réaliser la fusion

```
git merge rentree_suivante
```

... et là : **Merge conflict !**

```
Auto-merging cours.tex
CONFLICT (content): Merge conflict in cours.tex
Automatic merge failed; fix conflicts and then commit the result.
```

Certains paragraphes ont un contenu différent, il va falloir gérer ça à la main.



Gérer un merge conflict

Ouvrir le fichier `cours.tex`. git a modifié son contenu pour mettre en évidence le conflit.

```
<<<<<<< HEAD
    ancien contenu
=====
    nouveau contenu
>>>>>>> rentree_suivante
```

git status permet de voir sur quels fichiers il existe des conflits :

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:   cours.tex
#
```



Gérer un merge conflict

La démarche à suivre pour résoudre le conflit :

- corriger les conflits dans chaque fichier concerné
- vérifier que tous les conflits sont corrigés avec **git status**
- indexer (**git add**) tous les fichiers modifiés dans la manipulation
- commiter (**git commit**), on appelle ça un *'merge commit'*



Corriger des conflits dans un fichier plus facilement

Il est assez compliqué de s'en sortir avec ça :

```
<<<<<<< HEAD
  ancien contenu
=====
  nouveau contenu
>>>>>>> rentree_suivante
```

On va utiliser un outil graphique : **Meld** (il existe énormément d'équivalents)



Corriger des conflits dans un fichier plus facilement

```
git mergetool
```

mergetool va lancer Meld (ou équivalent) pour chaque fichier où il y a un conflit.



Utiliser Meld

Meld (et équivalents) présente sur trois colonnes, trois version du fichier :

- **locale** à gauche : votre dernière version de la branche réceptrice
- **remote** à droite : la dernière version de la branche à fusionner
- **origin** au centre : la dernière version commune aux deux branches

Dans la pratique, on enregistrera le résultat voulu dans la colonne centrale, c'est à dire sur **origin**.



Finaliser la résolution de conflits

mergetool créé un fichier de sauvegarde <fichier>.orig, on peut les supprimer une fois la session mergetool terminée

Il faut faire un *merge commit* : valider la résolution des conflits au sein d'un commit

```
git status
git add cours.tex
git commit -m "Merge dans master de rentrée_suivante"
```



Mettre en ligne votre branche

```
git push origin [ma_branche]
```



ma_branche

origin



Terminer une branche

```
git branch -d rentree_suivante
```

Il est possible de terminer seulement une branche qui a été mergée et n'a pas été modifiée depuis. Pour forcer la suppression d'une branche :

```
git branch -D rentree_suivante
```



Partie 4 : remonter le temps

C'est moi Doc! Je suis de retour du futur...



Accès à un commit

- **HEAD** : dernier commit de la branche courante
- **nom_branche/HEAD** : dernier commit de la branche nom_branche
- donner le nom d'une branche revient à pointer vers le dernier commit de cette branche

Il est possible de se "déplacer" relativement à un commit

- **HEAD^** : un commit avant HEAD
- **HEAD^^** : deux commits avant HEAD
- **HEAD~42** : quarante deux commits avant HEAD



Accès à un commit

- trouver l'identifiant court d'un commit

```
git log --oneline
```

- **master@{20/09/2012}** : prendre un commit à une date donnée



Voyager dans le temps

- mettre un fichier tel qu'il était à un commit donné

```
git checkout [nom_commit] [fichier]
```

Cette manipulation modifie directement le fichier dans votre répertoire de travail.

- remettre un fichier "en l'état"

```
git checkout HEAD [fichier]
```



Annexe 1 : Quelques conseils

tout va bien se passer...



Commitez bien, commitez souvent

- segmentez les tâches, faites un commit dès que possible
- ne commitez pas sciemment un code qui ne marche pas



Les branches c'est bon, mangez-en !



- séparez les tâches indépendantes
- n'hésitez pas à faire des expérimentations dans une branche dédiée
- gardez une branche "stable" à tout moment de votre développement



Jouez collectif

- organisez-vous et concertez-vous avec vos partenaires
- donnez des titres compréhensibles à vos commits
- suivez les avancées des autres, donnez votre avis
- construisez votre cycle de développement intelligemment : en fonction de la taille et la nature de votre équipe, vos deadlines, *etc.*



Annexe 2 : Guide de survie

Comme Rambo, mais en mieux.



Au début de la journée de travail

- je commence toujours par récupérer le travail des autres

```
git pull
```

- je vérifie sur quelle branche je me trouve

```
git branch  
git checkout
```



Quand je code

- j'ajoute mes fichiers modifiés à l'index

```
git add
```

- je vérifie mes modifications suivies

```
git status
```

- je valide mes changements dans un commit

```
git commit
```



Partager mon travail

- je publie mon travail au moins en fin de journée

```
git push
```

- en cas de fast-forward, j'applique mes modifications après celles des autres

```
git pull --rebase
```

- en cas de conflit, je fait les modications à la main

```
git mergetool  
git rebase --continue
```



Au secours, rien ne va plus !

- faire attention à ce qu'on fait pour éviter les ennuis inutiles
- DON'T PANIC !
- prendre le temps de lire les messages d'erreur de git (ils donnent souvent la solution)
- il existe de très nombreuses ressources sur le net



Des questions ?

Ne mourrons pas idiots.



Merci de votre participation
et à une prochaine fois!

