Initiation à GIT et à Netbeans

FIP 1ere année
2014-2015

Avertissement

- Ce texte contient plein de franglais absolument affreux, et j'assume
  - un, c'est fatigant de chercher des équivalents pour tout
  - deux, traduire certains termes techniques — qui restent en anglais dans les noms de commandes, me semble peu productif, voire gênant quand on voudra chercher des explications complémentaires sur le web
- par exemple, pour « check out », la « meilleure » traduction en contexte que je vois serait « relever »
  - ça n'aide pas beaucoup à comprendre
  - il vaut mieux utiliser le terme anglais pour chercher sur le web

Netbeans

- Notion d’IDE
- Notion de projet
- Comment organiser vos projets sur le disque dur
  - **IMPORTANT**: ne pas imbriquer vos projets
- Manipulations de base

Logiciel de gestion de versions

- Problèmes:
  - travail sur du code: on veut pouvoir éventuellement revenir en arrière en cas d'erreur et garder l'historique des modifications
  - travail **collectif** : plusieurs développeurs travaillent sur le même programme
    - problème de cohérence (ils peuvent modifier le même fichier),
    - problème de récupération du travail des autres
    - gestion de différentes versions du logiciel
- Tout une série de programmes pour ça : CVS, SVN**, Git**, Mercurial

Logiciels centralisés/ décentralisés

- **Centralisés:** il faut un serveur central, qui contient l'archive du logiciel. Les utilisateurs font des copies de cette archive, qu'ils doivent synchroniser quand ils font des modifications. Exemple : CVS, SVN

- **Décentralisés:** chaque utilisateur a sa propre version de l'archive, qu'il peut gérer lui-même. La synchronisation peut se faire d'utilisateur à utilisateur, ou sur un serveur. Exemple : Git, Mercurial

SVN (Subversion)

- fonctionne en ligne de commande ou avec des interfaces graphiques
- pour chaque projet, on a une archive principale, soit dans un dossier du disque, soit sur un serveur
- certains utilisateurs (pour l'instant, chacun d'entre vous a son compte) peuvent accéder à une archive SVN

GIT

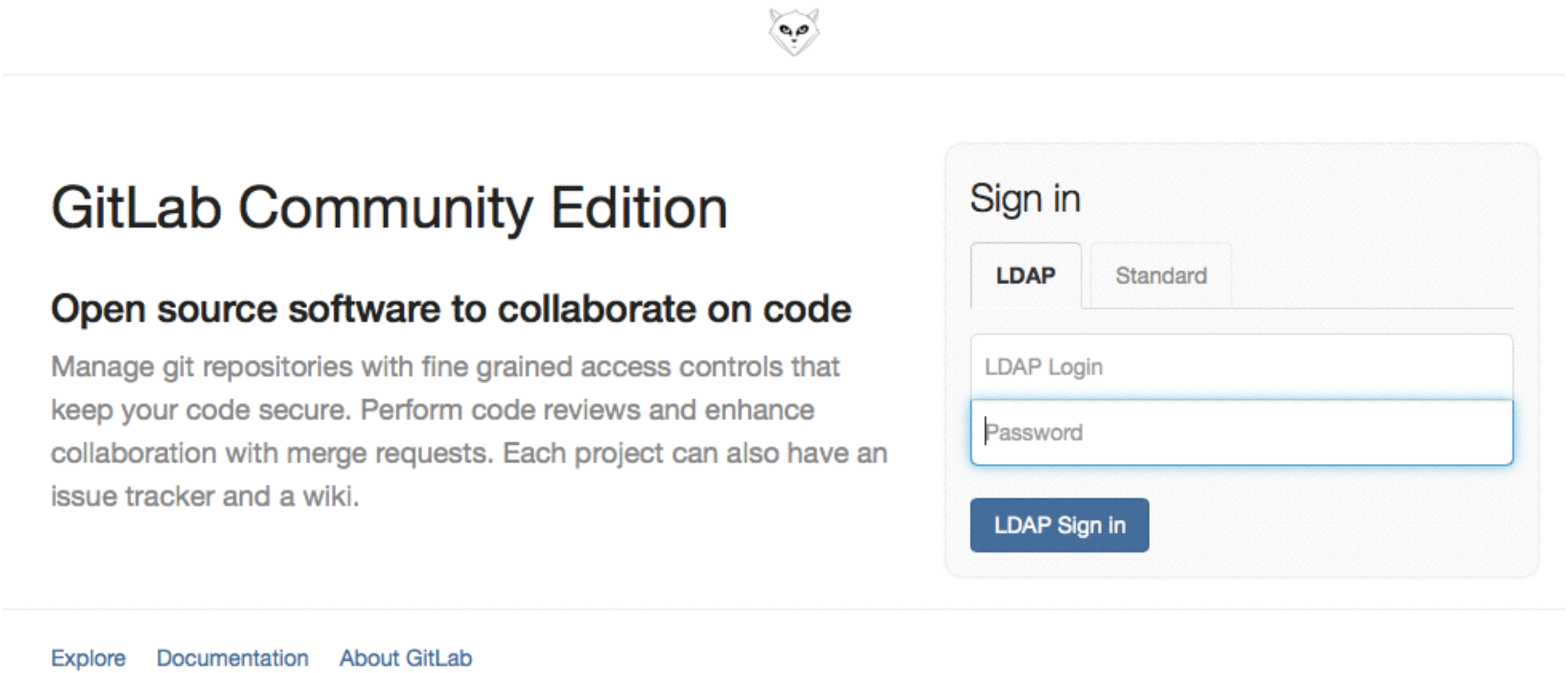
- Système décentralisé de gestion de version développé pour le noyau linux par Linus Torvalds himself

- décentralisé :
  - chaque utilisateur a sa propre version de l’archive
  - pas besoin de serveur (mais peut exister)
  - plusieurs méthodes de synchronisation

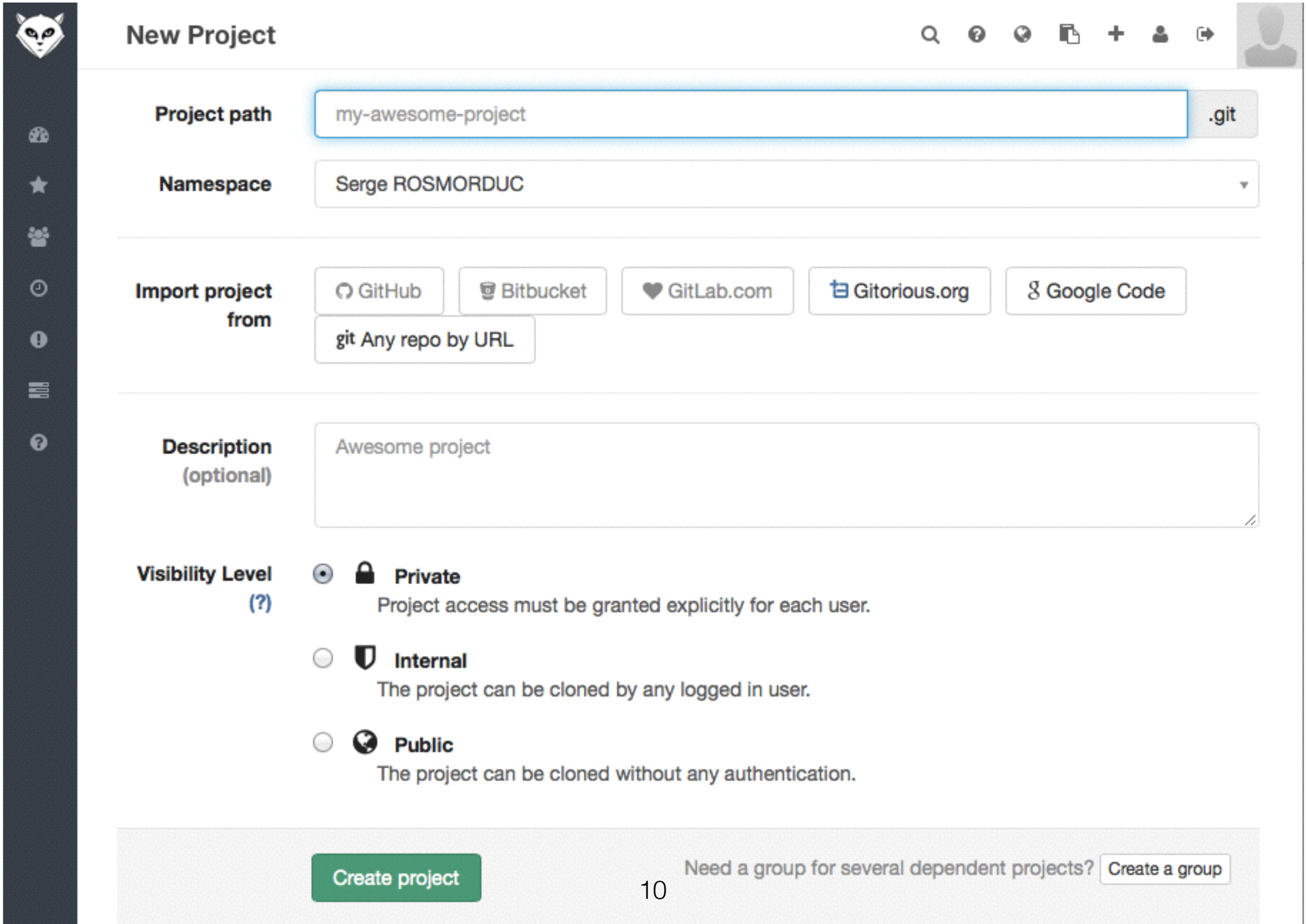
GITLAB

- Serveur pouvant héberger des projets Git
- Vous y avez un compte
- Vous pouvez créer des projets, les partager éventuellement avec vos

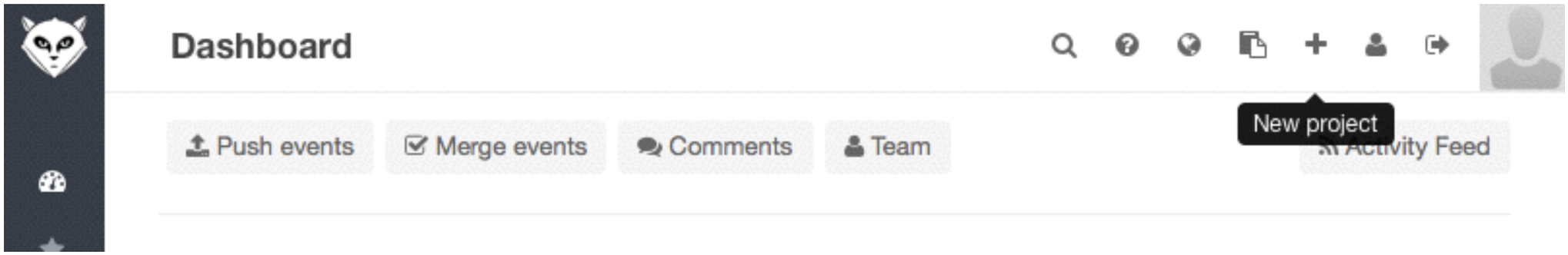
Connexion

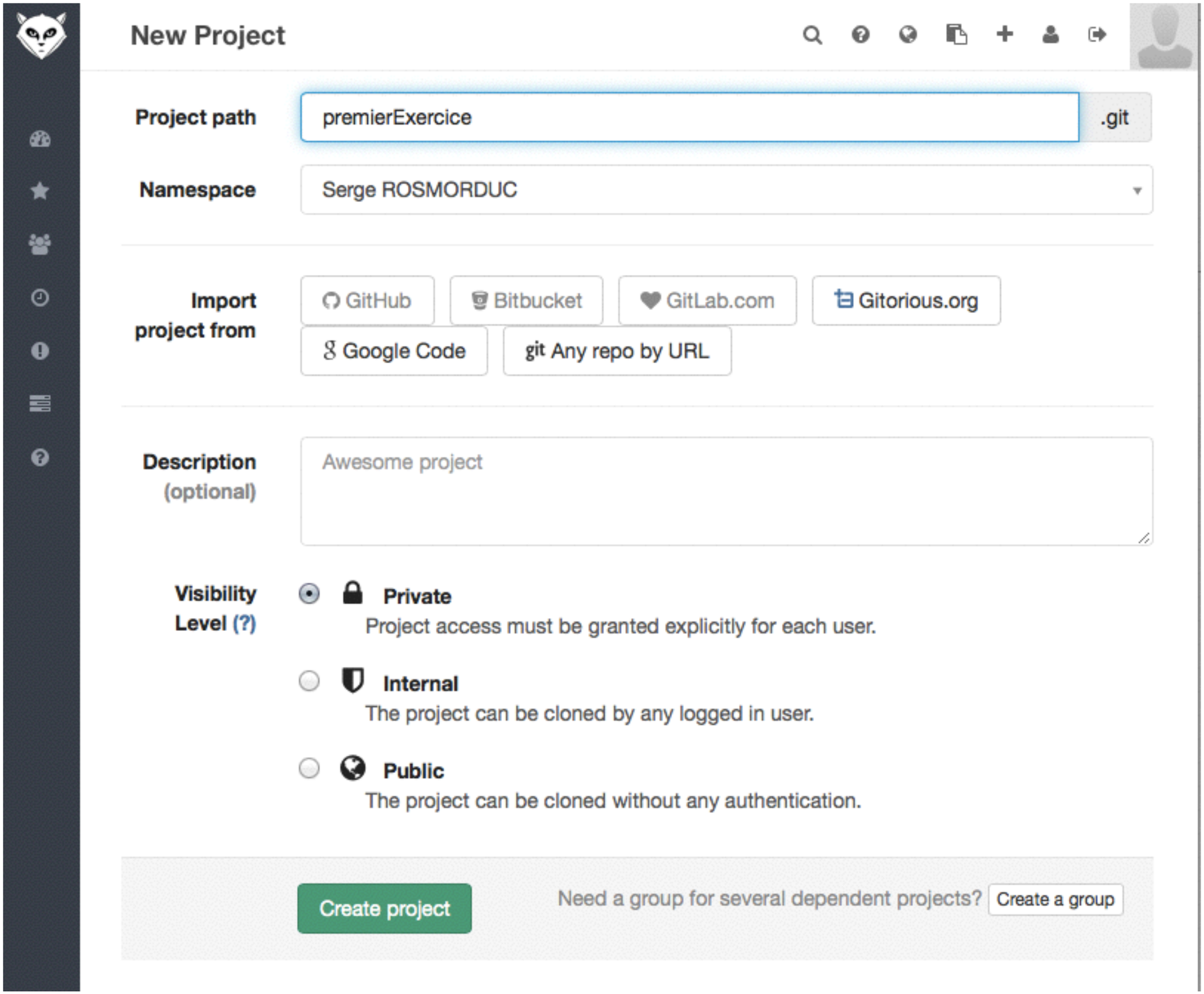

GitLab Community Edition
Open source software to collaborate on code
Manage git repositories with fine grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.
Sign in
LDAP
Standard
LDAP Login
Password
LDAP Sign in
Explore
Documentation
About GitLab

Page des projets



Création de projet




New Project
Project path
premierExercice
.git
Namespace
Serge ROSMORDUC
Import project from
GitHub
Bitbucket
GitLab.com
Gitorious.org
Google Code
git Any repo by URL
Description (optional)
Awesome project
Visibility Level (?)
Private
Project access must be granted explicitly for each user.
Internal
The project can be cloned by any logged in user.
Public
The project can be cloned without any authentication.
Create project
Need a group for several dependent projects?
Create a group

Serge ROSMORDUC / premierE...

Project was successfully created.

P – Edit

Star 0

The repository for this project is empty

You can add a file or do a push via the command line.

SSH HTTP [redacted]cnam.fr:[redacted]/premierExercice.git private

Command line instructions

Git global setup

Command line instructions

Git global setup

```
git config --global user.name "Serge ROSMORDUC"
git config --global user.email "[redacted]"
```

Create a new repository

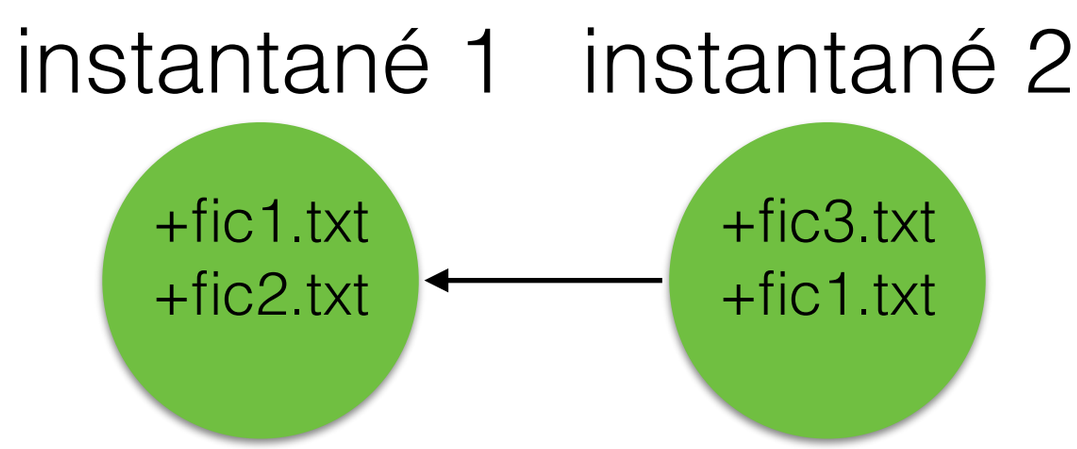
```
git clone [redacted]cnam.fr:[redacted]/premierExercice.git
cd premierExercice
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Existing folder or Git repository

```
cd existing_folder
git init
git remote add origin [redacted].cnam.fr:[redacted]/premierExercice.git
git push -u origin master
```

Fondamentaux de git

- Une archive git contient un graphe d'instantanés
- un « instantané » est un ensemble de
  - suppressions de fichiers
  - créations/nouvelles versions de fichiers
- Si j'ai le graphe suivant



- l'application de tous les instantanés me donnera trois fichiers (fic1.txt, fic2.txt, fic3.txt), et fic1.txt aura le contenu qu'il a dans l'instantané 2 (effectué après le 1)

Cycle de vie des fichiers

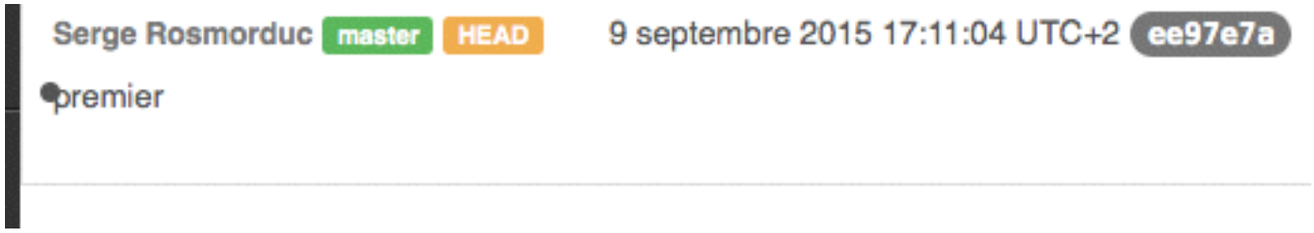
- On a des informations sur un fichier à différents endroits:
- l'historique, qui contient les instantanés (je parlerai désormais de commits)
- l'index, qui contient la liste des fichiers qui seront inclus dans le prochain commit.

➜ un fichier modifié peut ne pas être inclus dans un commit si on le désire

- le répertoire de travail, dans lequel on peut voir/modifier/supprimer des fichiers

Un exemple simple

- dans un dossier géré par git:
  - création de fic1.txt
  - ajout à l'index:
    - git add fic1.tx
  - commit



Ajout de deux fichier

second commit

premier commit

master branche

HEAD position actuelle

HEAD

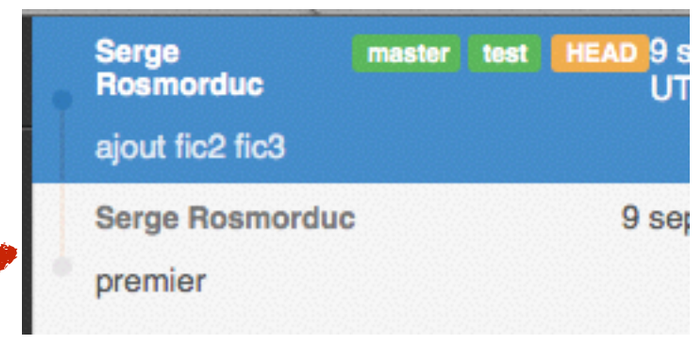
- position actuelle sur l'archive
- est modifiée par les commits et par la commande checkout
- git checkout ee97 (nom du premier commit)
- ls … plus que fic1.txt !
- git checkout master. HEAD revient à la fin de master

Les branches

- On travaille (presque) toujours dans une branche
- git branch: affiche la branche en cours
- c’est elle qui est modifiée par les commits, etc…
- une branche est définie par sa tête (le « pointeur » en vert qu’on a vu précédemment)

branches

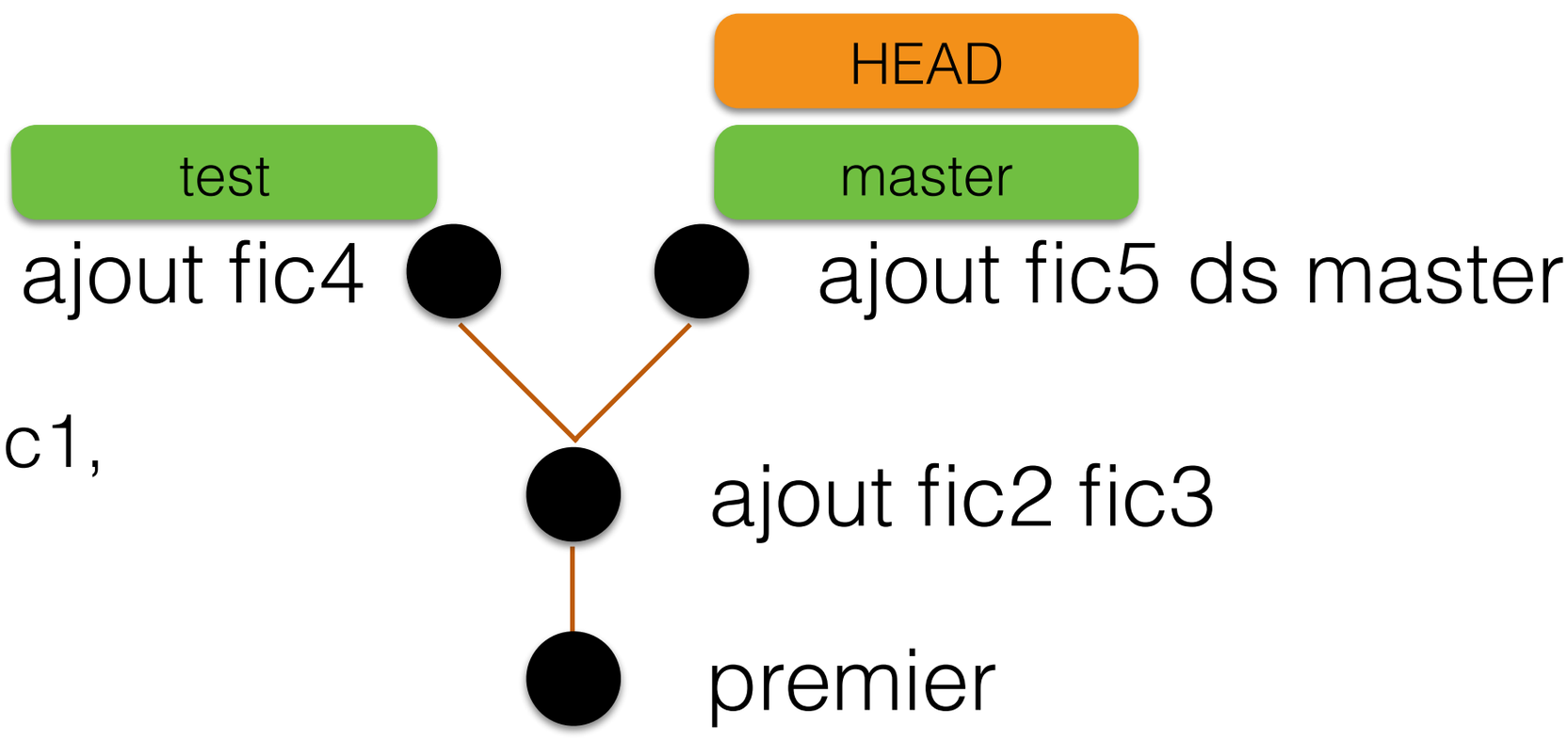
- git branch test
- git checkout test
- add et commit…




```
localhost:demo rosmord$
localhost:demo rosmord$ git branch
* master
  test
localhost:demo rosmord$ git checkout test
Switched to branch 'test'
```

branches…

- on repasse dans « master »
- git checkout master
- modifications et commit dans master…
- master contient: fic1, fic2, fic3 et fic5
- test contient fic1, fic2, fic3 et fic4

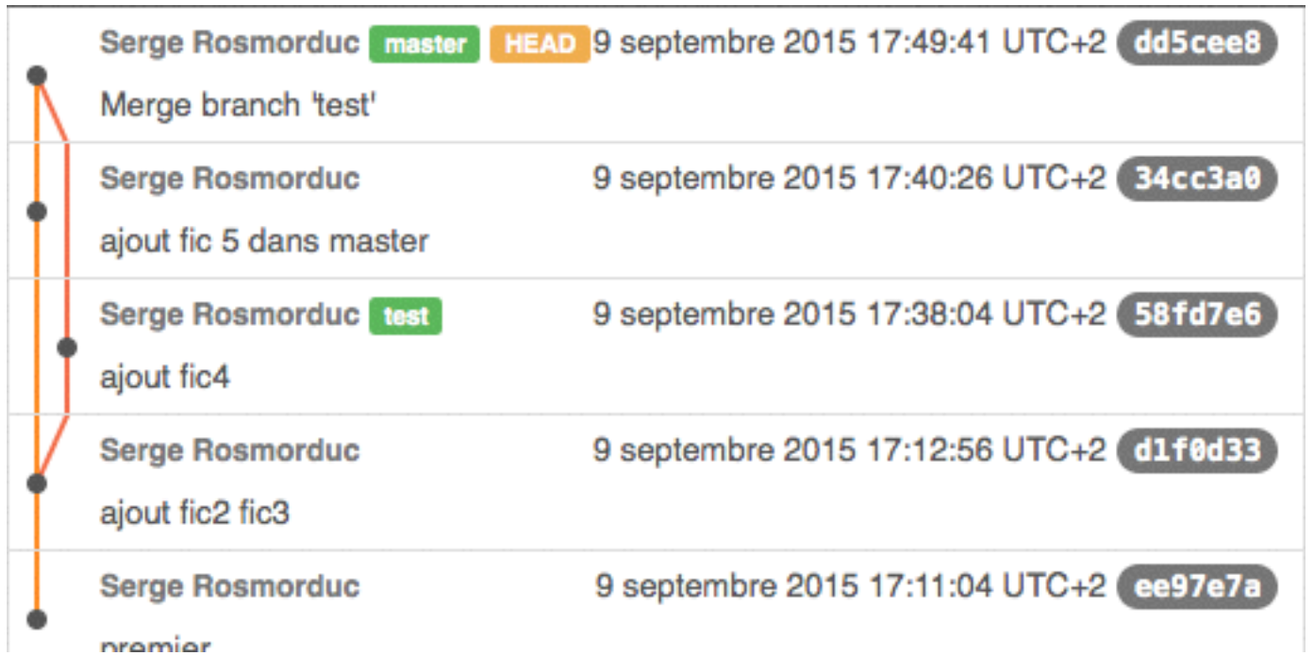


merge

- quand on veut réunir deux branches
- se placer (checkout) dans la branche *à modifier*
- faire un merge en donnant le nom de la branche avec laquelle on merge

git checkout master

git merge test



detached head

- on checkout un ancien commit
- git checkout ee97
- si on effectue des modification, elles ont lieu dans une branche anonyme
- « detached head »
- pour les conserver, il faut nommer la branche
- git checkout -b maBranche
- sinon, elle disparaîtra quand on changera de branche

Conflit

- dans une branche dev, on a modifié fic1.txt
- dans master, on a aussi modifié fic1.txt
- on merge:

```
localhost:demo rosmord$ git branch
  dev1
* master
  test
localhost:demo rosmord$ git merge dev1
Auto-merging fic1.txt
CONFLICT (content): Merge conflict in fic1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

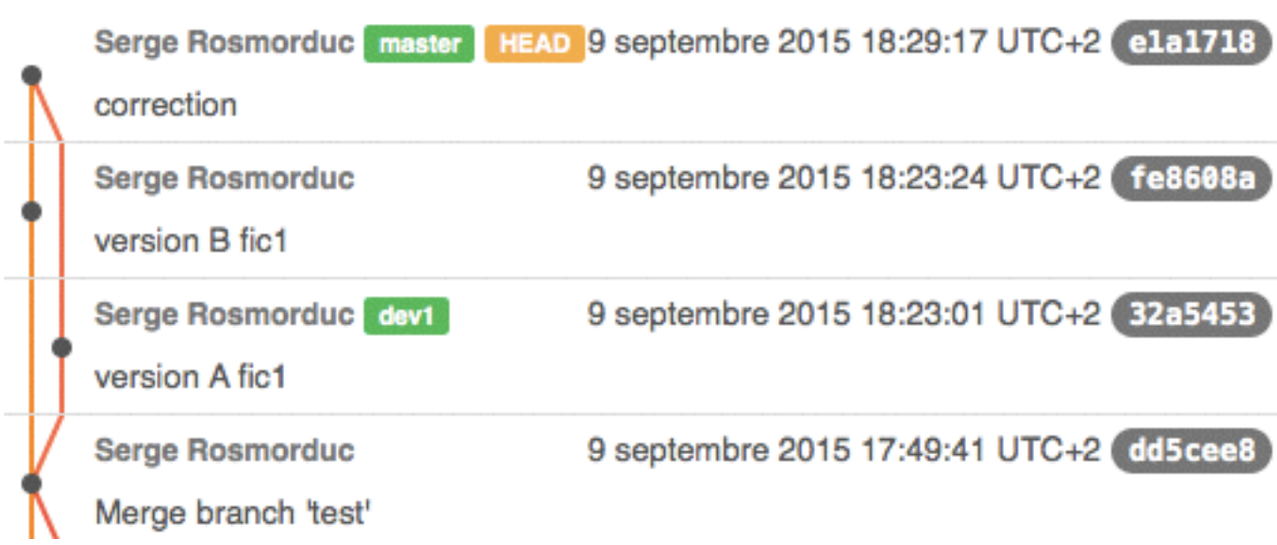
Conflit

- On ouvre fic1.txt
- on le modifie
- on le sauve…
- on peut enfin faire un commit…

```
<<<<<<< HEAD
toto
=======
titi
>>>>>>> dev1
```

version de master

version de dev1



Conflit de destruction d’un fichier

- un fichier a été modifié dans une branche, et détruit dans une autre.
- dans ce cas, le merge échoue, et on dit
  - si on veut conserver le fichier :
    - git add leFichier.txt
  - si on veut le détruire
    - git rm leFichier.txt
  - le commit terminera le merge

Avec un serveur…

- Trois opérations pour échanger des modifications entre plusieurs archives git
- peuvent s’essayer en local, sans serveur…
- initialisation d’un répertoire git (une archive centralisée) :
  - git init —bare nomArchive.git
- ensuite…

Échange de données

- **git clone ARCHIVE DESTINATION**: pour initialiser un clone d'une archive
- **git push :** envoie les dernières mises à jour vers l'archive d'origine
- **git fetch** : récupère les dernière mises à jour depuis l'archive d'origine
- **git fetch NOM_ARCHIVE :** récupère les mises à jour faites dans l'archive dont on fournit le chemin

Échange de données

- git **push** et git **fetch** réussissent **toujours**
- elles créent de nouvelles branches
- pas de conflits (nouvelles branches)
- après un fetch, on fait *souvent* un merge pour intégrer les nouvelles modifications
- fetch + merge = pull
- ***git pull*** *est un raccourci pour git fetch suivi de git merge*

Git et Netbeans

- Deux approches:

  A. un projet git contient plusieurs projets netbeans

     - nous l’utiliserons pour les tps

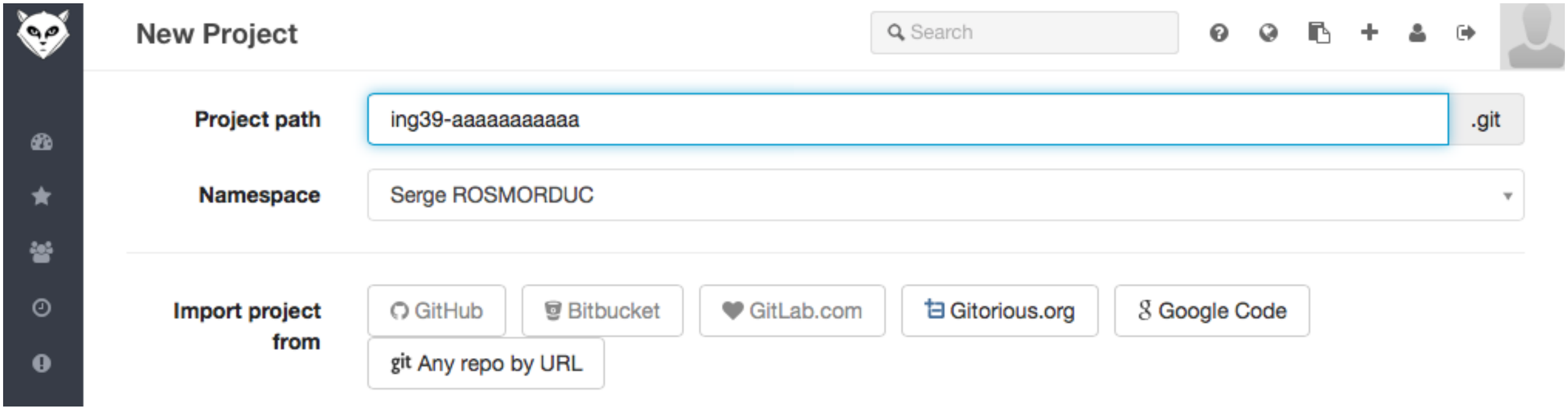
  B. un projet git = un projet netbeans

     - nous l’utiliserons pour les devoirs collaboratifs (plusieurs étudiants sur la même archive)

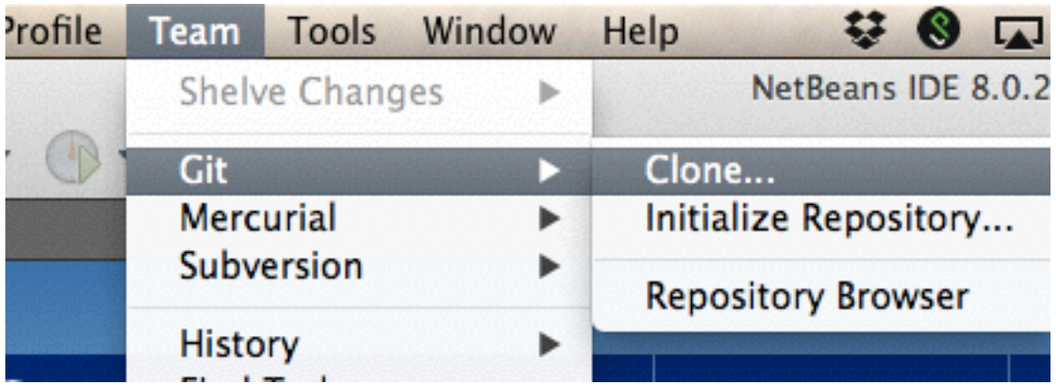
Un projet git pour plusieurs projets netbeans

- création d’un projet git
- Team/git/remote/clone du projet dans netbeans
- création de projets netbeans dans le dossier du projet git.
- commit, etc..

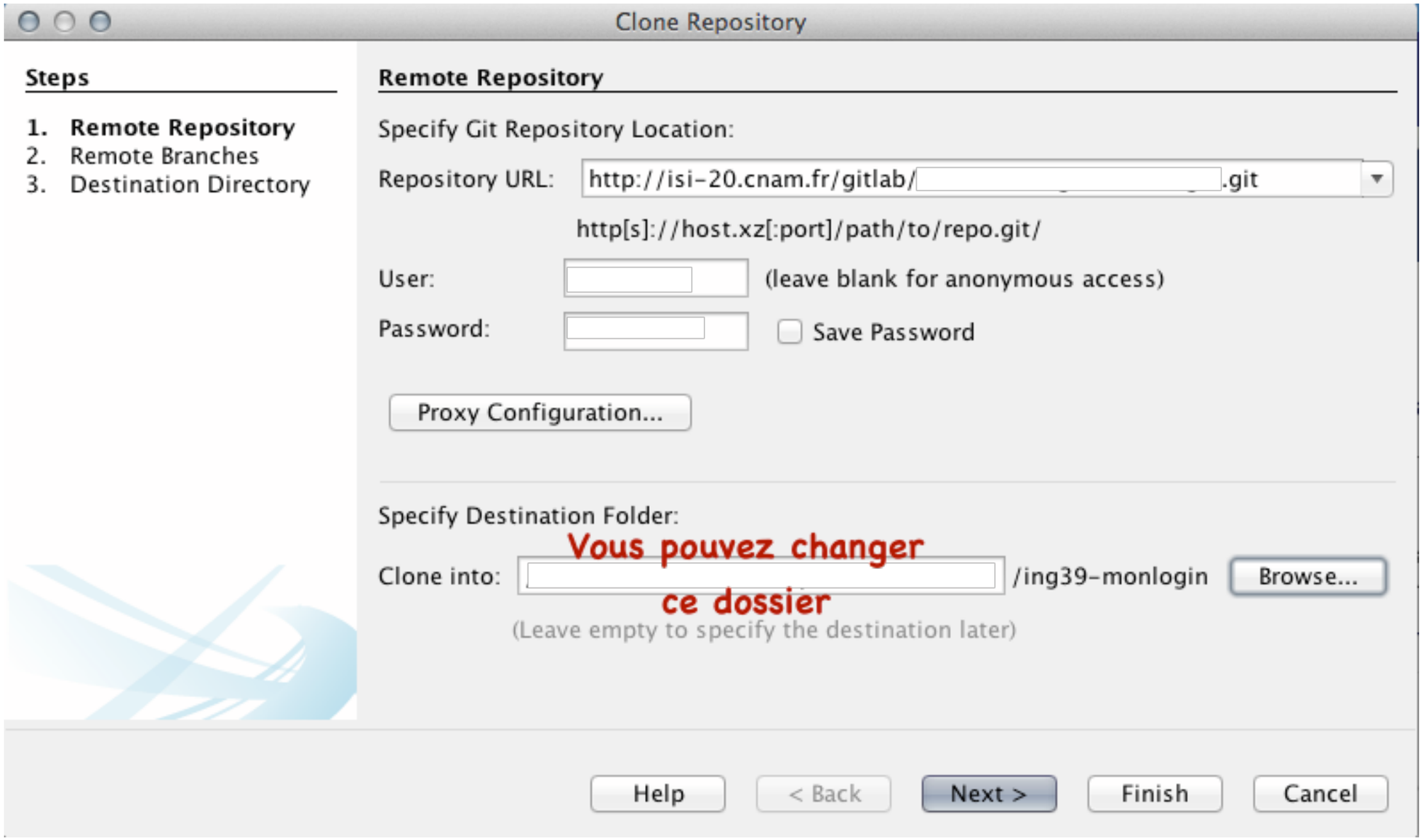
Création d'un projet git sur la forge



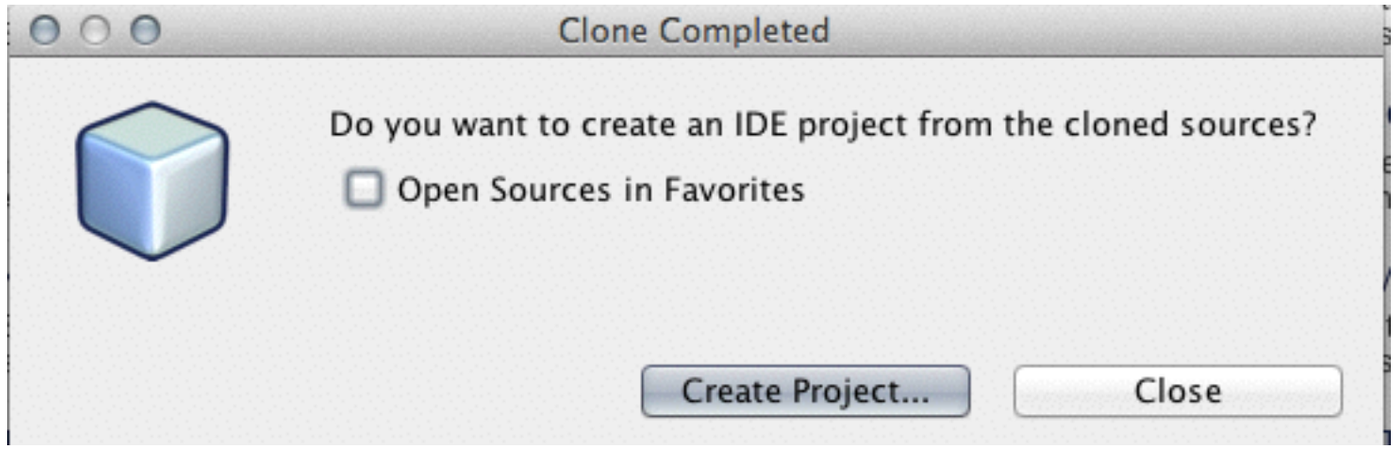
Clone (Team/git/clone)

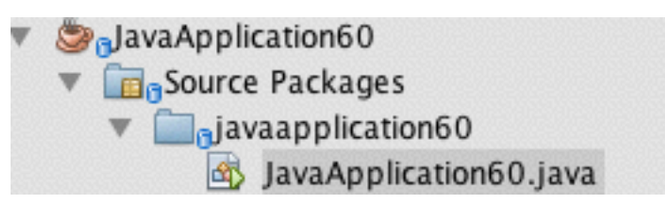


Clone (Team/git/clone)



Création d'un projet netbeans

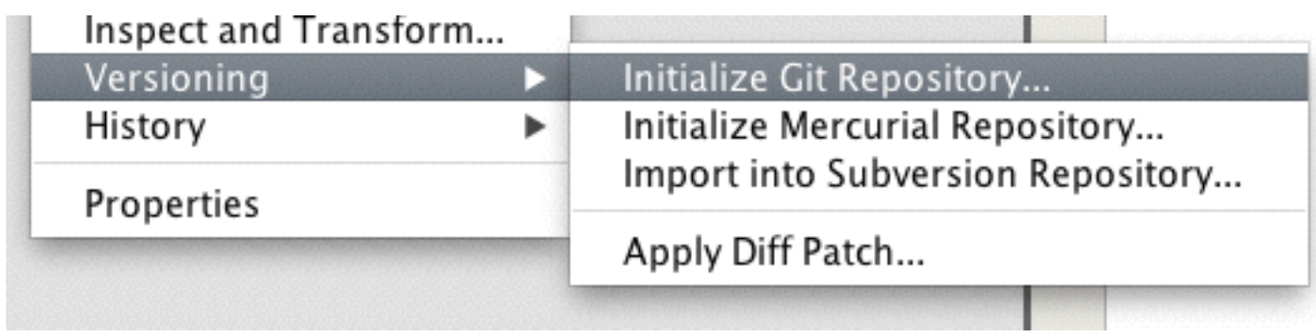




Un projet git = un projet netbeans

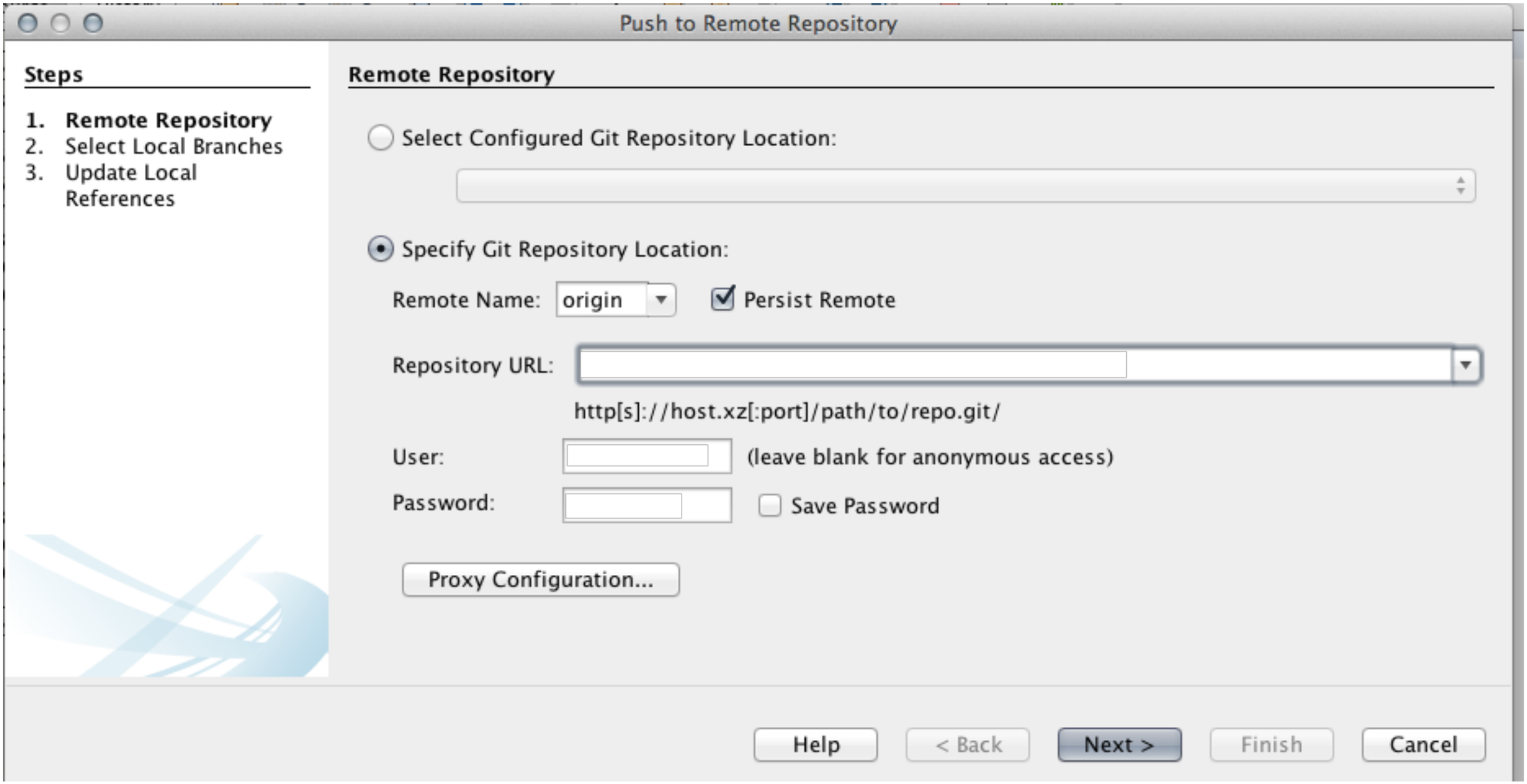
- le plus simple: créer un projet netbeans ***(en dehors d'une archive git)***

- en faire une archive git locale:



- commit

- création d’un projet vide sur la forge
- **push** du projet netbeans vers la forge



Travailler avec un projet

- plusieurs étapes:
  - commit : sauver dans l'archive **locale** les modifications effectuées
  - push: envoyer les modifications de l'archive locale vers le serveur
- en théorie:
  - on travaille, on compile, on « commite »
  - quand c'est un peu stable, on « push »
- en pratique ici: commit + push systématique

Commit initial

Commit – JavaApplication61

Commit Message:

premier commit

Author: Serge Rosmorduc <serge.rosr

Commiter: Serge Rosmorduc <serge.rosr

Amend Last Commit

Files to Commit:

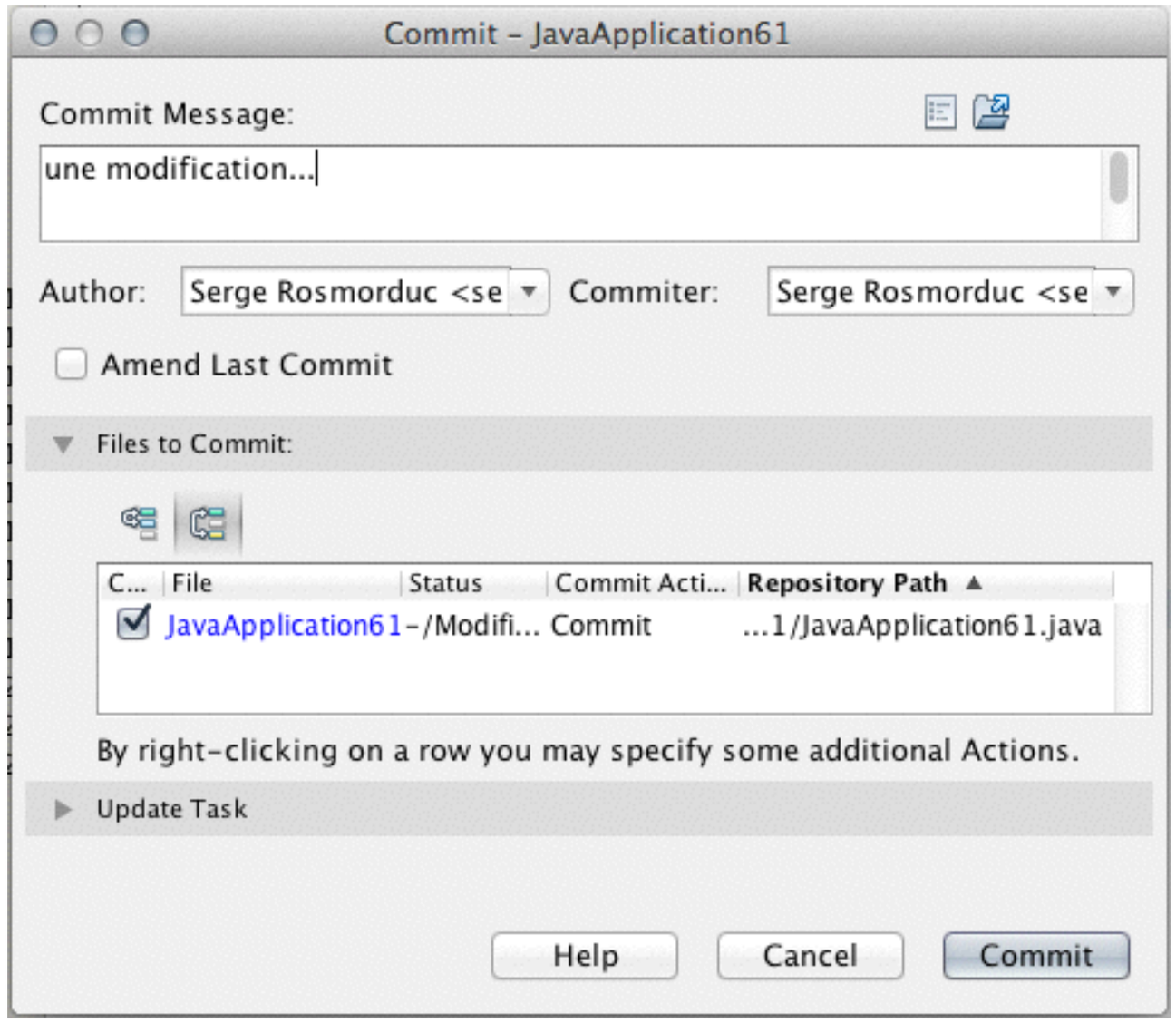
C...	File	Status	Commit Action	Repository Path
☑	build.xml	-/Added	Commit	JavaApplication61/build.xml
☑	manifest.mf	-/Added	Commit	JavaApplication61/manifest.mf
☑	build-impl.xml	-/Added	Commit	...on61/nbproject/build-impl.xml
☑	genfiles.properties	-/Added	Commit	...1/nbproject/genfiles.properties
☑	project.properties	-/Added	Commit	...1/nbproject/project.properties
☑	project.xml	-/Added	Commit	...cation61/nbproject/project.xml
☑	JavaApplication61.jav	-/Added	Commit	...ation61/JavaApplication61.java

By right-clicking on a row you may specify some additional Actions.

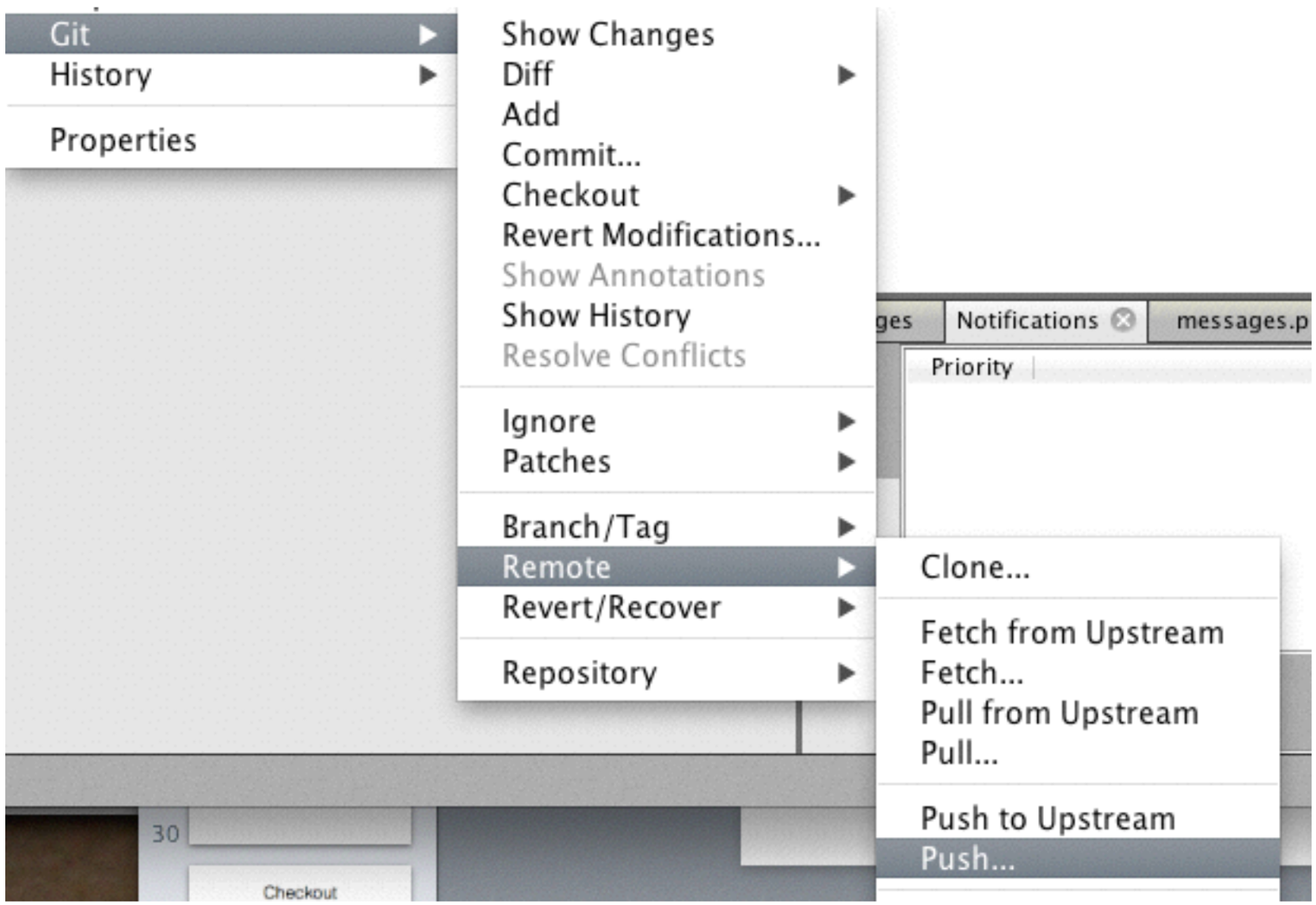
Update Task

Help Cancel Commit

Commit après modification…



Push


Git
History
Properties
Show Changes
Diff
Add
Commit...
Checkout
Revert Modifications...
Show Annotations
Show History
Resolve Conflicts
Ignore
Patches
Branch/Tag
Remote
Revert/Recover
Repository
Clone...
Fetch from Upstream
Fetch...
Pull from Upstream
Pull...
Push to Upstream
Push...
Notifications
messages.p
Priority
30
Checkout

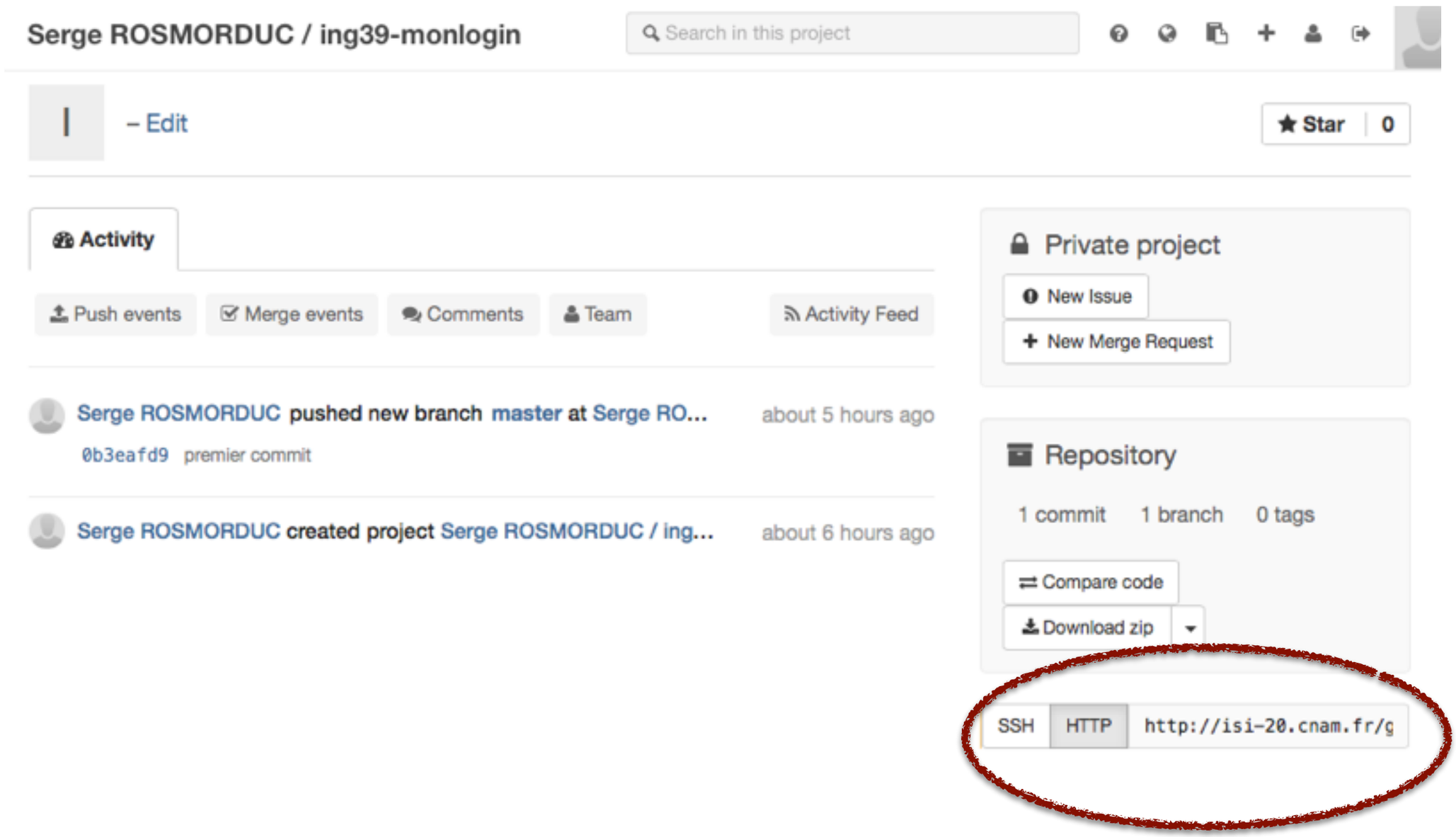
Important

SANS PUSH ➜ PAS DE SAUVEGARDE SUR LE SERVEUR

COMMIT + PUSH !!!!!!

Clone

- Sur une nouvelle machine ou autre utilisateur
- donner l'adresse du projet
- faire un clone avec cette adresse



Une dernière commande : pull

- scenario: vous avez modifié le projet depuis chez vous
- vous voulez que le projet actuellement sur votre compte au CNAM soit mis à jour (et récupère les modifications)
- pull= met à jour le projet à partir du serveur Git
- pull= fetch + update : récupération des différence, et application de celles-ci. Parfois, on sépare les deux
- dans tous les cas: merge fusionne la version distante et la version locale.

Conseils

- pour ce cours, faites des « commit/push » assez fréquemment: vous risquerez moins de perdre votre travail

- dans la vraie vie, quand on travaille à plusieurs:

  - on push du code qui tourne (pour que les autres n'aient pas à le débugger), sauf si on travaille sur des branches différentes.

Conflits

- Si deux utilisateurs modifient le même fichier dans un projet
- pas de problème pour le commit : se fait sur une branche locale, ni pour le fetch
- problème lors du **merge : quel fichier choisir ?**

Conflits

- à l'origine…

```
*/
public class JavaApplication61 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Première version.");
    }

}
```

Conflit (1)

- machine 1

```java
/**
 *
 * @author rosmord
 */
public class JavaApplication61 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Version machine 1.");
    }

}
```

Conflit (2)

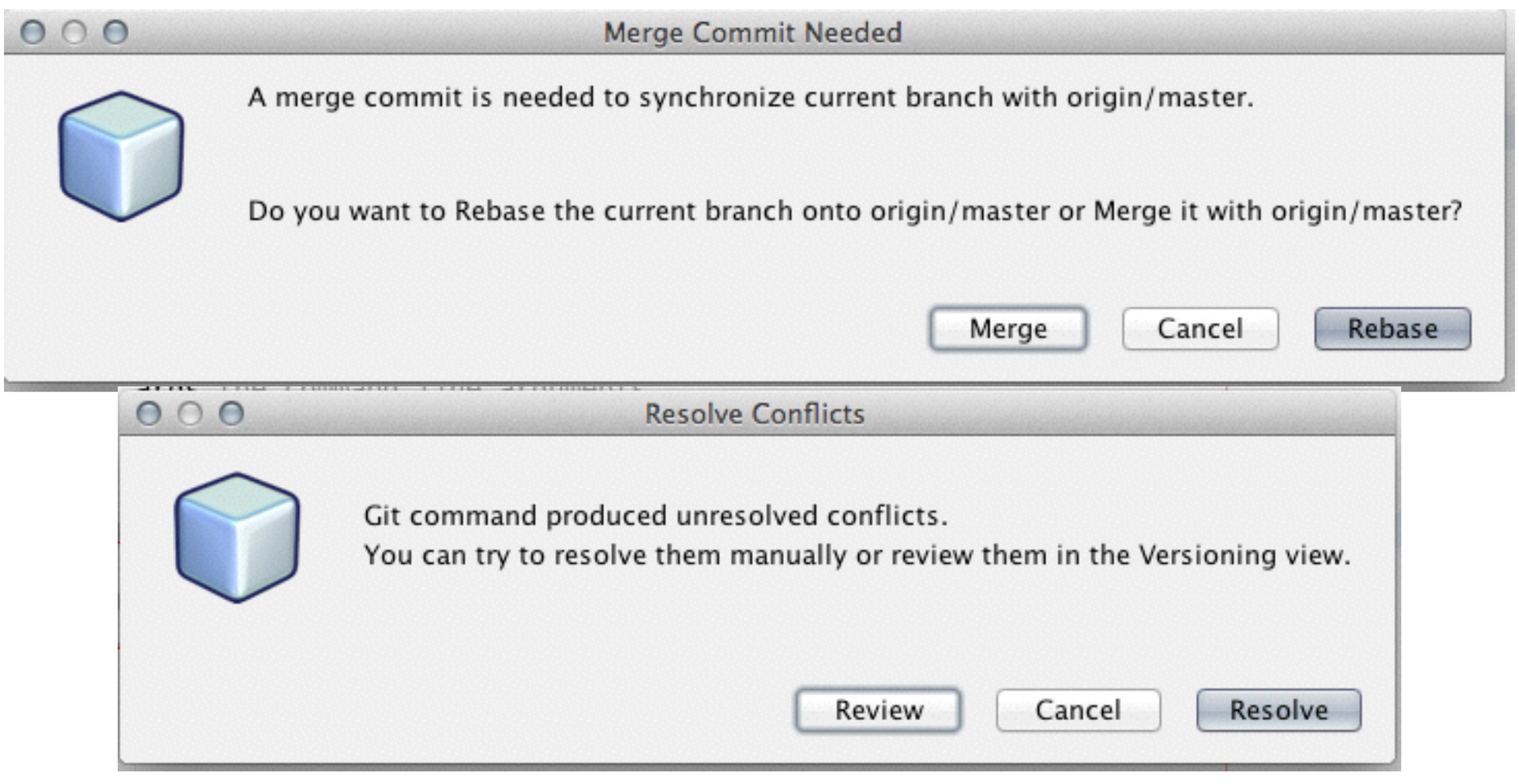
- Modification, commit et push sur machine 2

JavaApplication61.java 446 Bytes

```java
/*
 * To change this license header, choose License Headers
 * To change this template file, choose Tools | Template
 * and open the template in the editor.
 */
package javaapplication61;

/**
 *
 * @author rosmord
 */
public class JavaApplication61 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Version machine 2.");
    }

}
```

Conflit (3)

- On essaie de faire un pull sur machine 1



Conflit (4)

Conflict: 1 of 1, Unresolved: 1

Accept | Accept Both | Accept & Next

```
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
18*        System.out.println("Version machine 1");
19     }
20
21 }
```

HEAD (9415604) – Merge origin/master

Accept | Accept Both | Accept & Next

```
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
18*        System.out.println("Version machine 2")
19     }
20
21 }
```

origin/master (83264b2) – machine 2

```
14     /**
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
 *
18     }
19
20 }
```

Result after merge

JavaApplication61.java

OK | Cancel | Help

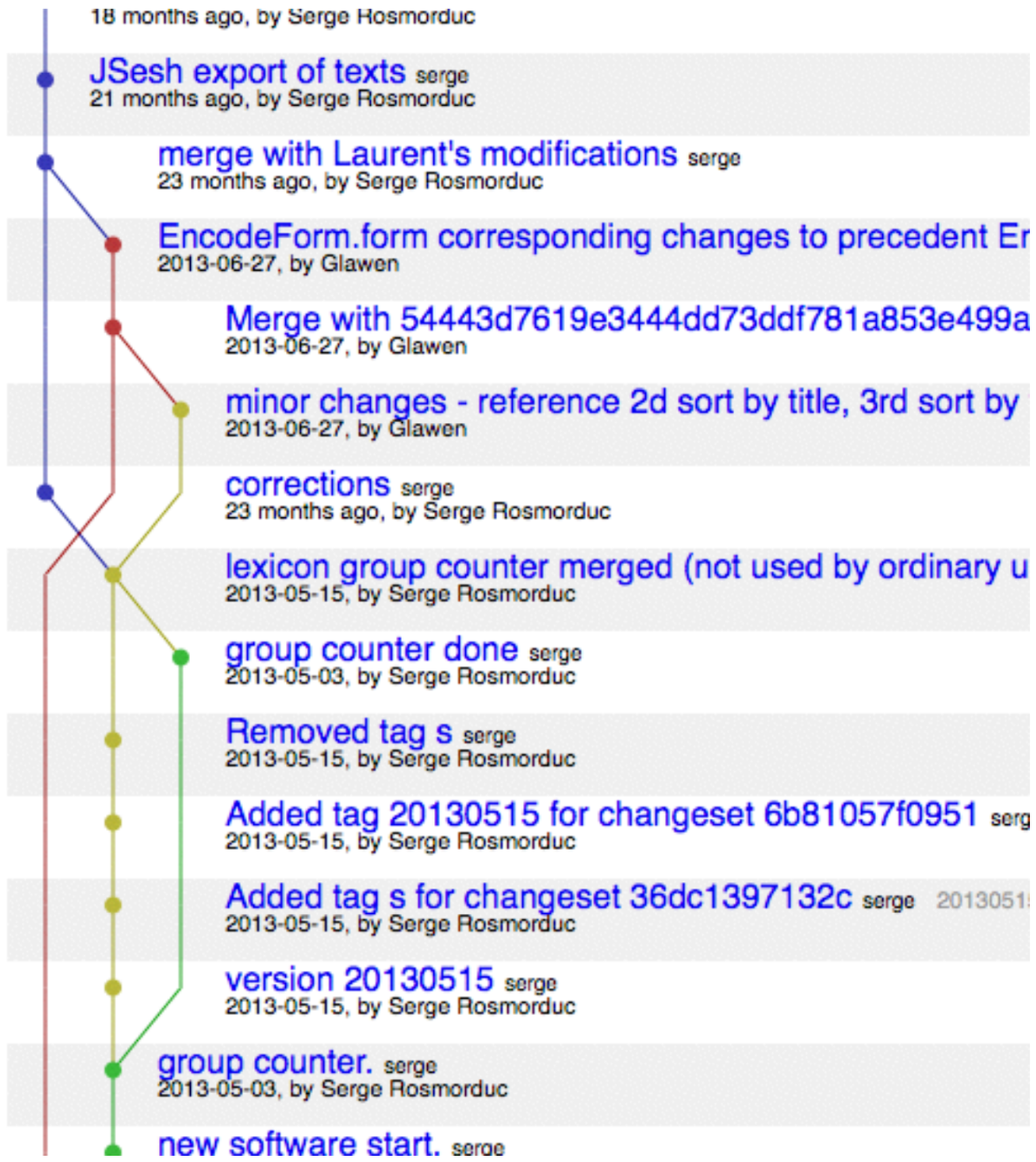
Conflit… commit + push…

```
10  * @author rosmord
11  */
12 public class JavaApplication61 {
13
14     /**
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
18         System.out.println("Version machine 2");
19         System.out.println("Version machine 1");
20     }
21
22 }
```

branches

- possibilités d’avoir plusieurs branches de développement
  - jointes avec merge ou rebase
- développement en parallèle, version stable/version expérimentale, etc…

branches


18 months ago, by Serge Rosmorduc
JSesh export of texts serge
21 months ago, by Serge Rosmorduc
merge with Laurent's modifications serge
23 months ago, by Serge Rosmorduc
EncodeForm.form corresponding changes to precedent Er
2013-06-27, by Glawen
Merge with 54443d7619e3444dd73ddf781a853e499a
2013-06-27, by Glawen
minor changes - reference 2d sort by title, 3rd sort by
2013-06-27, by Glawen
corrections serge
23 months ago, by Serge Rosmorduc
lexicon group counter merged (not used by ordinary u
2013-05-15, by Serge Rosmorduc
group counter done serge
2013-05-03, by Serge Rosmorduc
Removed tag s serge
2013-05-15, by Serge Rosmorduc
Added tag 20130515 for changeset 6b81057f0951 serg
2013-05-15, by Serge Rosmorduc
Added tag s for changeset 36dc1397132c serge 2013051
2013-05-15, by Serge Rosmorduc
version 20130515 serge
2013-05-15, by Serge Rosmorduc
group counter. serge
2013-05-03, by Serge Rosmorduc
new software start. serge

Compléments (en ligne de commande)

- Dans un projet Git, un fichier peut être dans plusieurs états:
  - archivé : il a été commité
  - modifié : il a été modifié par rapport à l'archive
  - planifié (staged) : il a été modifié, puis ajouté à la liste des fichiers qui seront inclus dans le prochain commit

Exemple

- on a un projet avec deux fichiers archivés, toto1.txt et toto2.txt

Exemple

- on modifie toto1.txt, et on crée un nouveau fichier, toto3.txt.

- on tape: **git status**

```
# 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:   toto1.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  toto3.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Exemple

```
# 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:   toto1.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  toto3.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

- toto1.txt n'a pas été ajouté au commit, mais est tracé
- toto3.txt n'est pas tracé (ni donc ajouté au commit)
- git commit -a -m « test » ferait un commit de toto1.txt uniquement
- git add toto3.txt suivit du commit précédent fera un commit de la totalité du code.

Ooops!

- On a fait des modifications, des « add », et on veut revenir à l'état du dernier commit

git reset —hard . : annule toute modification depuis le dernier commit

git clean -df . : supprime les fichiers créés depuis le dernier commit (sauf ceux listés dans .gitignore).

Webographie

- « become a git guru » : https://www.atlassian.com/git/tutorials/ : très clair, avec des images :-)

- Scott Chacon et Ben Straub, *Pro Git :* https://git-scm.com/book/fr/v1 livre disponible gratuitement (et en français!)

- Une référence visuelle de Git: https://marklodato.github.io/visual-git-guide/index-fr.html plein de petits dessins vous expliquant graphiquement les commandes de git.