IRISA

Éléments de Génie Logiciel

Cours 4

—

Gestion de Version

—

Jean-Marc Jézéquel
(d'après Thomas Genêt)
IRISA/ISTIC - Université de Rennes
e-mail : jezequel@irisa.fr
http://people.irisa.fr/Jean-Marc.Jezequel

istic Informatique Électronique

Université de Rennes

Déroulement du TP 1-2 par rapport au CMM ?

Capability Maturity Model = évaluation du processus de développement

Niveau 1, Initial/Chaotique/Heroïque

- Développement dans l'urgence, non reproductible
- Le succès du projet dépend d'un petit nombre d'individus
- Comportement du logiciel non prédictible (partiellement testé)

Objectif de l'UE GEN : passer au Niveau 2 :

Niveau 2, Reproductible/Discipliné

- Sauvegarde, partage, restauration projet, gestion de version (CM4)
- Comportement du logiciel prédictible, défauts détectés (test systématique : CM5)
- Assurance qualité du logiciel, documentation (CM6)
- Tâches définies pour les développeurs, qui fait quoi ? (CM7)

Il reste 3 niveaux au dessus ☺ : défini, maîtrisé, optimisant

Pourquoi la gestion de version ?

1. Développer collaborativement un code commun, en évitant

2. Sauvegarder le code, pour qu'il survive à

3. Revenir à une version antérieure du code, si la dernière version fait

Gestion de versions... quel est le problème ?

Il était une fois Jo et Mo qui travaillaient sur le même projet...

- Ils copient sur leur machine le projet qui ressemble à ça : 

- Mo modifie le fichier A : 

- Jo ajoute un fichier B : 

- Comment font ils pour obtenir chacun une copie de 

?
- Combien de copies entre Jo et Mo sont nécessaires (au min.) ?
... et s'il y avait eu N développeurs et N modifications ?

Gestion de versions... ça se complique, Mo est une truffe !

Mo s'est trompé et souhaite annuler sa modification de A

- A partir de 

- Comment font ils pour obtenir chacun une copie de 
 ?
- Combien de copies entre Jo et Mo sont nécessaires (au min.) ?
- ... et pour N développeurs ?

Remarque 1 (Ca n'est possible que si au moins un développeur...)

- *garde un historique des projets qu'il copie*
- *sait quels sont les fichiers/répertoires à ajouter/supprimer d'une version à l'autre*

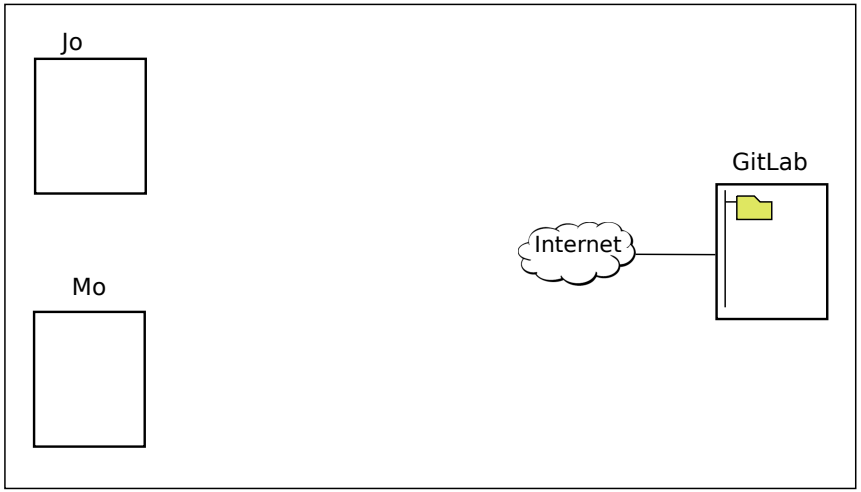
C'est précisément ce que fait un gestionnaire de versions, y compris sur les fichiers eux-mêmes : (suppression, ajout, modification de ligne)

Gestion de versions avec Git, le principe

- Chaque développeur à un **dépôt Git local** qui enregistre toutes les modifications opérées sur **son** projet (toutes les versions)
- Les développeurs qui le souhaitent peuvent synchroniser leur **dépôt local** avec un **dépôt partagé** (sur le GitLab ISTIC pour nous)
- Les développeurs interagissent avec Git avec les commandes :
  - créer le dépôt local : `init`
  - signaler des modifications/ajouts sur des fichiers/répertoires : `add`
  - supprimer un fichier : `rm` , supprimer un répertoire : `rm -r`
  - propager les modif./ajouts/suppressions sur le dépôt Git local : `commit`
  - propager les modif./ajouts/suppr. du Git local vers le GitLab : `push`
  - obtenir une **copie locale** d'un dépôt GitLab : `clone`
  - obtenir les dernières modifications faites sur le GitLab : `pull`

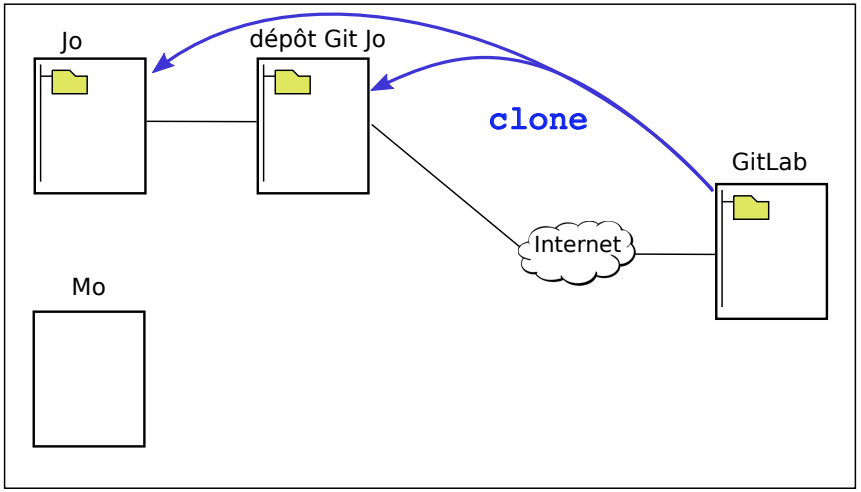
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo crée un nouveau projet (vide) sur GitLab



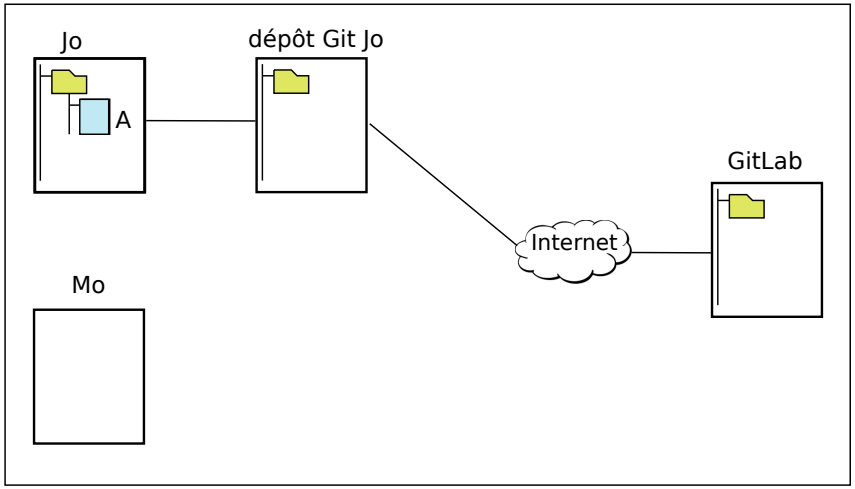
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo fait une copie (synchronisée) du projet provenant du GitLab



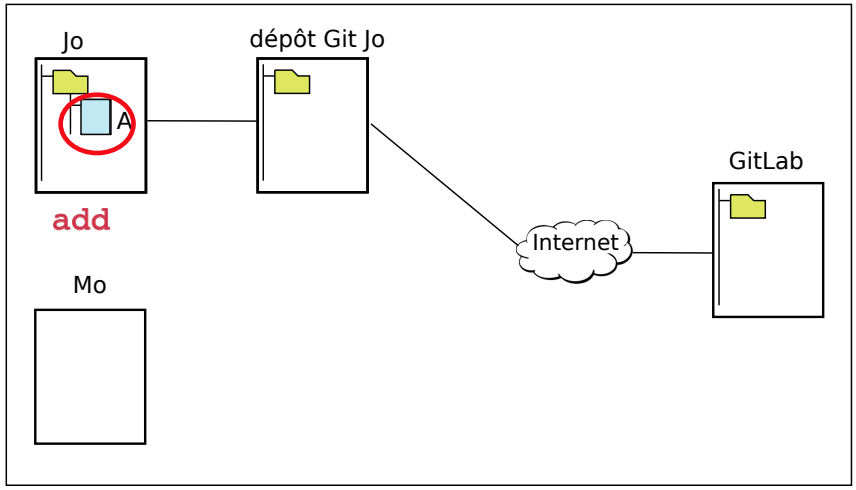
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo ajoute un nouveau fichier A dans sa copie du projet



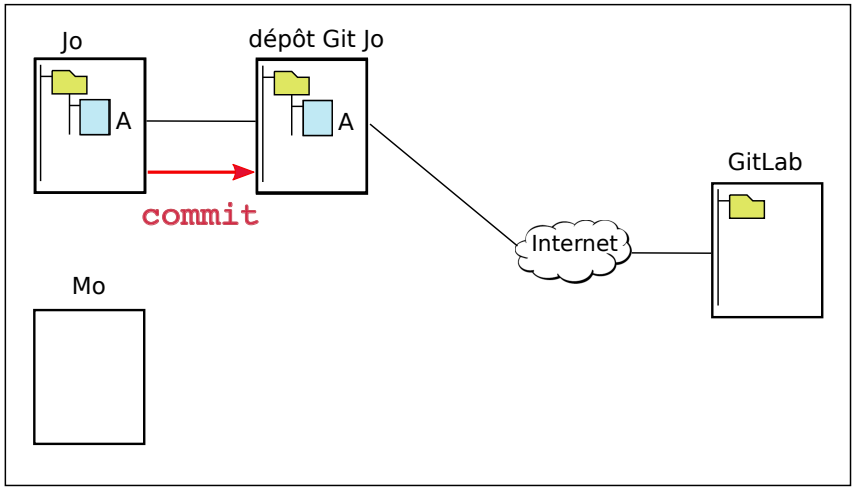
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo indique les fichiers/répertoires à ajouter à Git (a.k.a. staging)



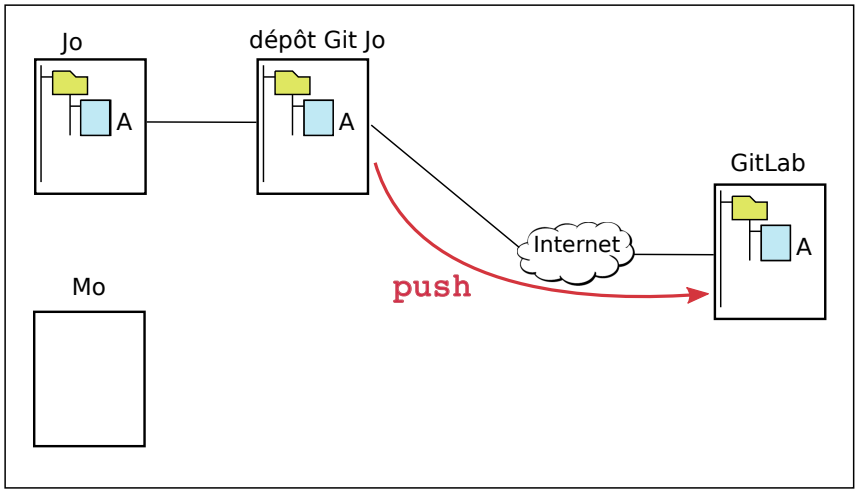
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo envoie les nouveaux fichiers/répertoires vers **son** dépôt Git



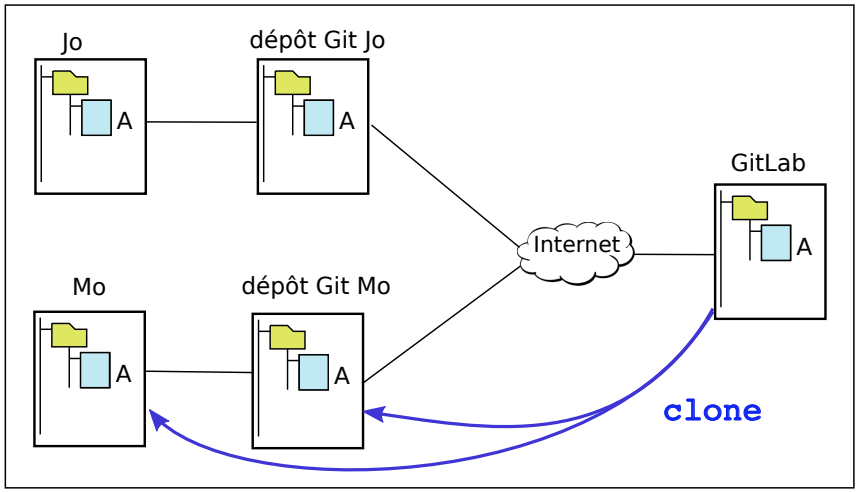
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo pousse ses modifications vers le GitLab



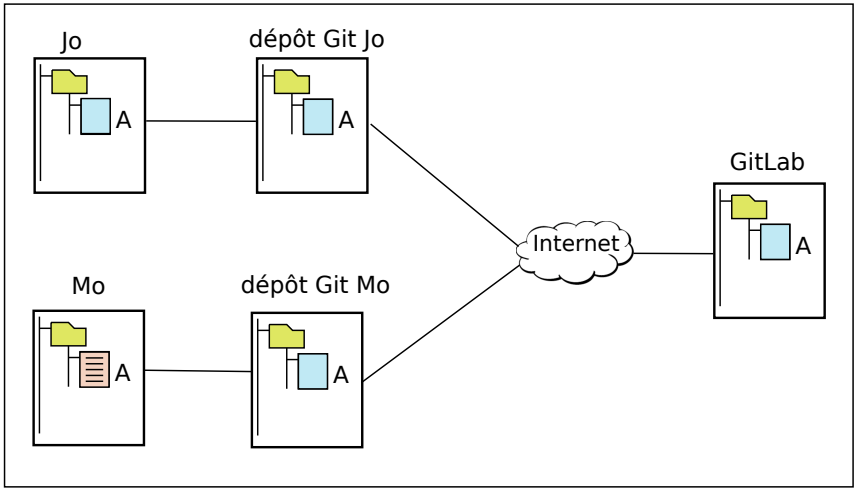
Il était une fois... Jo et Mo avec Git (l'initialisation)

Mo fait une copie (synchronisée) du dépôt sur le GitLab



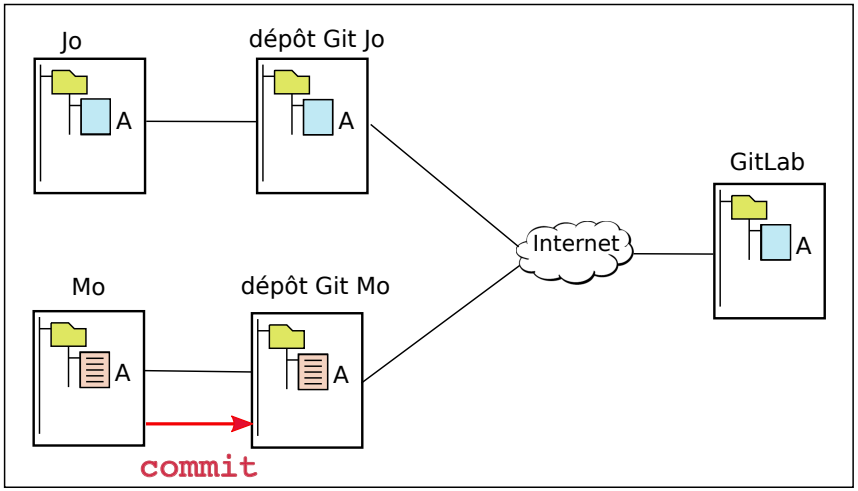
Il était une fois... Jo et Mo avec Git (utilisation)

Si Mo modifie le fichier A et veut envoyer la modification vers GitLab



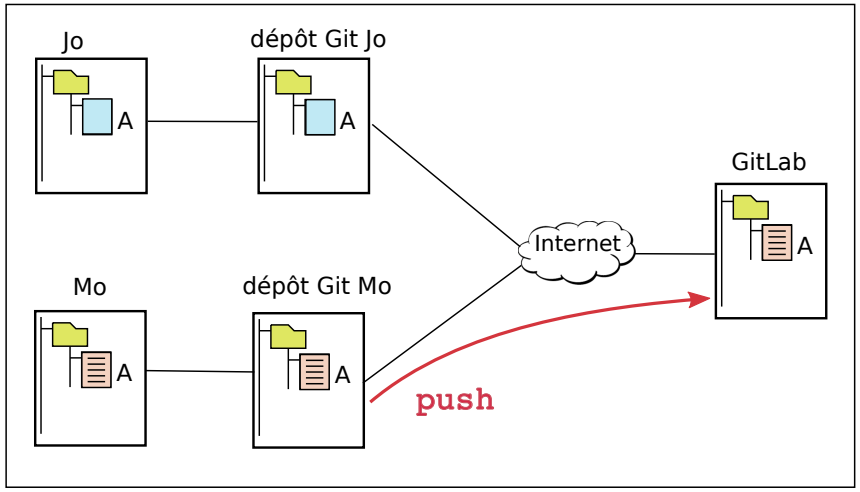
Il était une fois... Jo et Mo avec Git (utilisation)

Mo envoie les fichiers/répertoires modifiés vers **son** dépôt Git



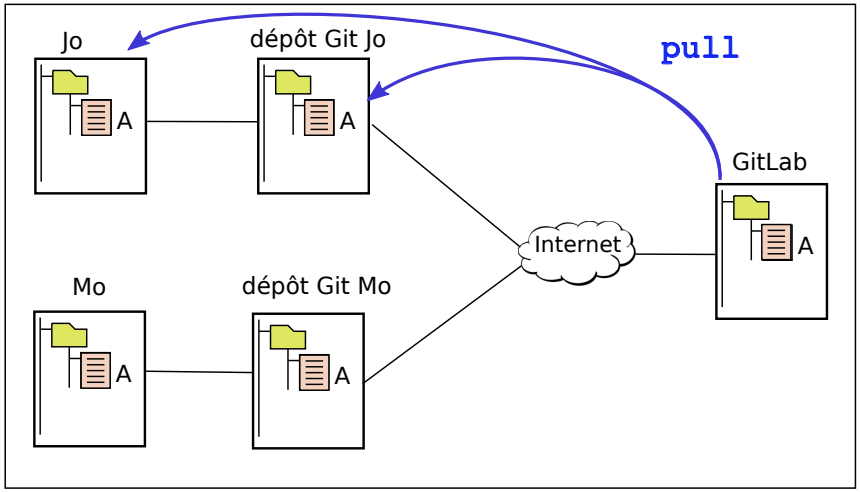
Il était une fois... Jo et Mo avec Git (utilisation)

Mo pousse ses modifications vers le GitLab



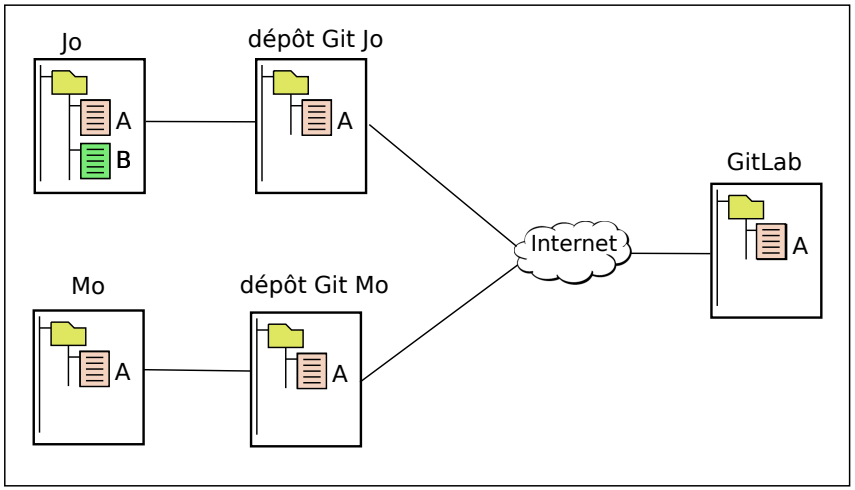
Il était une fois... Jo et Mo avec Git (utilisation)

Jo tire/applique les modifications du GitLab vers son dépôt et répertoire



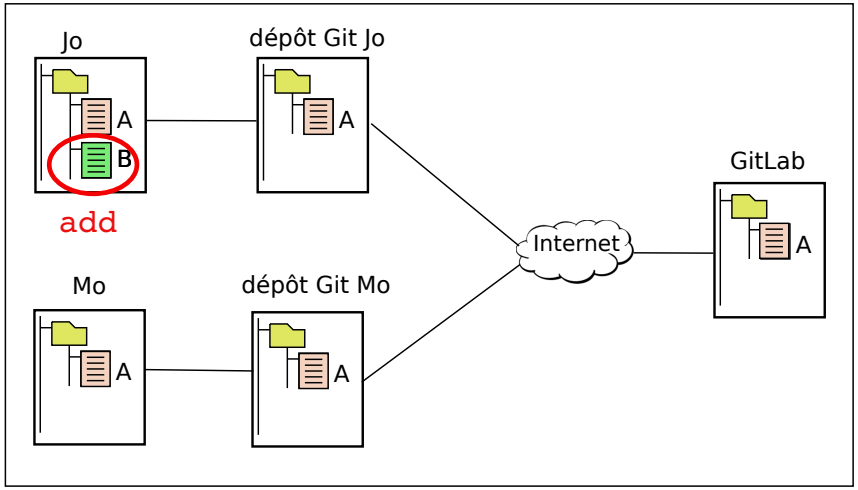
Il était une fois... Jo et Mo avec Git (utilisation)

Si Jo ajoute un fichier B et veut envoyer la modification vers GitLab



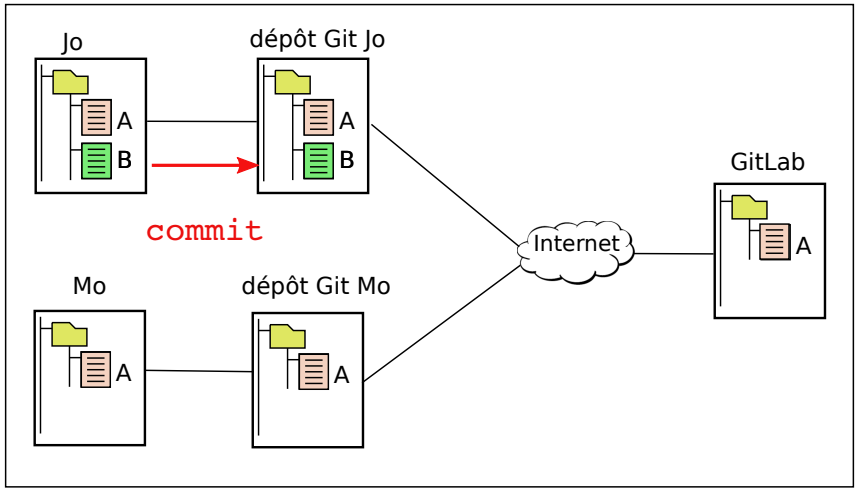
Il était une fois... Jo et Mo avec Git (utilisation)

Jo indique quels sont les fichiers à propager vers Git (a.k.a. staging)



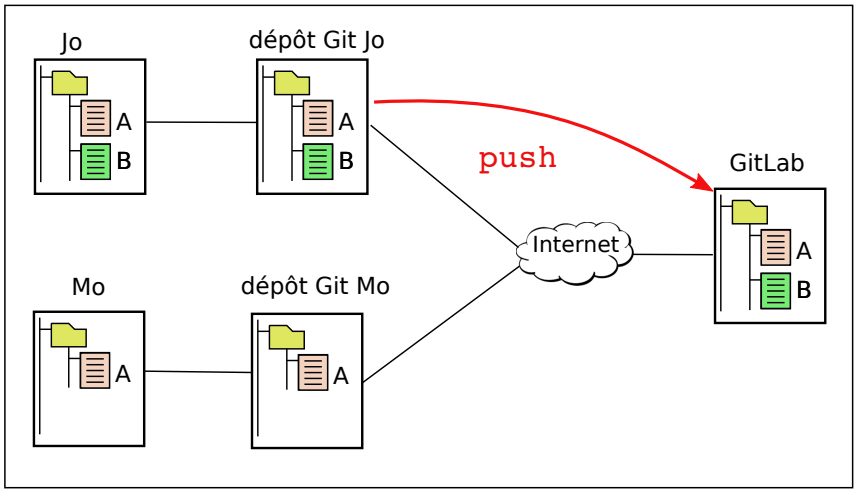
Il était une fois... Jo et Mo avec Git (utilisation)

Jo envoie le fichier supplémentaire vers **son** dépôt Git



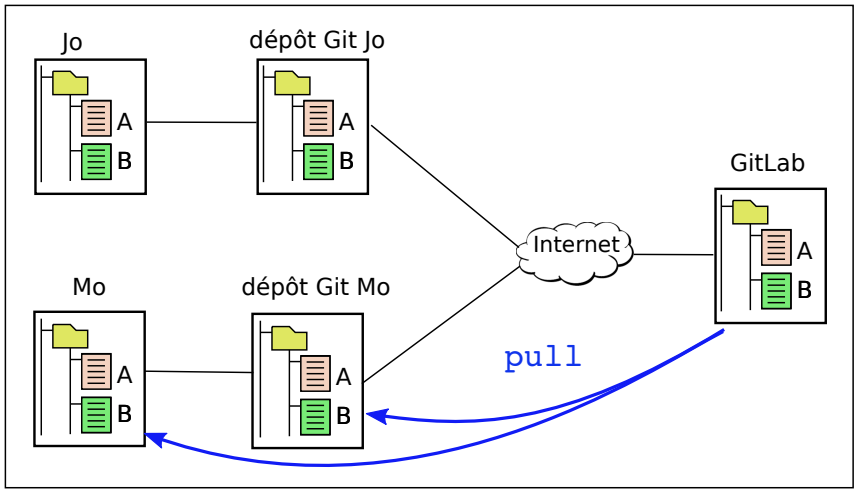
Il était une fois... Jo et Mo avec Git (utilisation)

Jo pousse ses modifications vers le GitLab



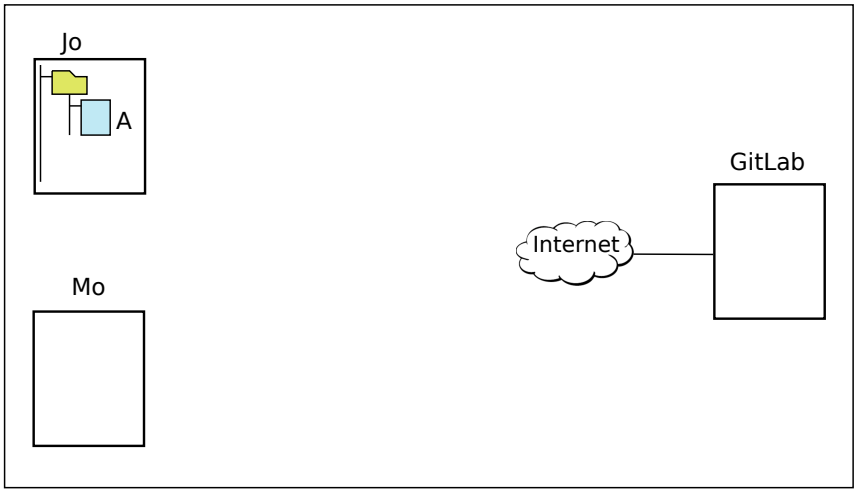
Il était une fois... Jo et Mo avec Git (utilisation)

Mo tire/applique les modifications du GitLab vers son dépôt et répertoire



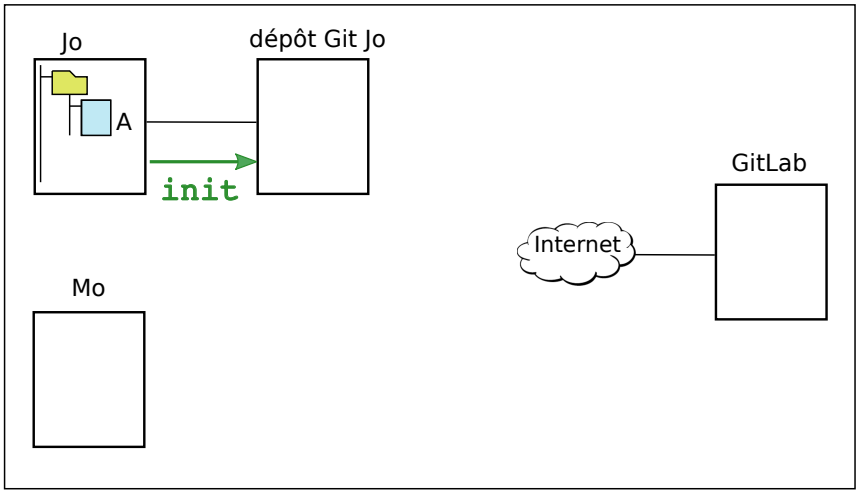
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo souhaite partager le projet contenu dans son répertoire



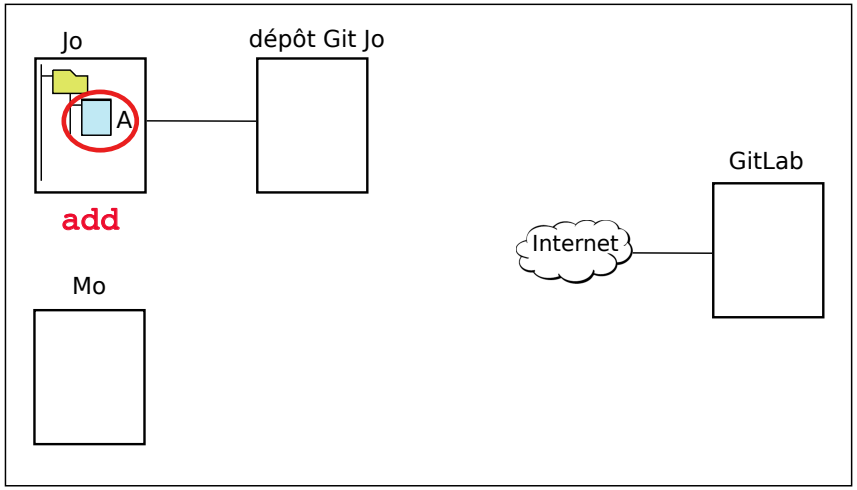
Jo et Mo avec Git (l'initialisation, une autre méthode)

L'initialisation du dépôt de Jo



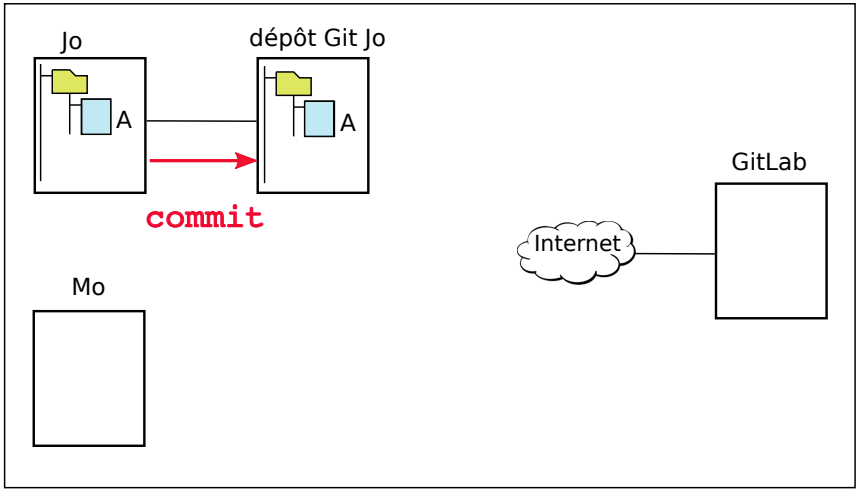
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo indique quels sont les fichiers/répertoires à ajouter Git (a.k.a. staging)



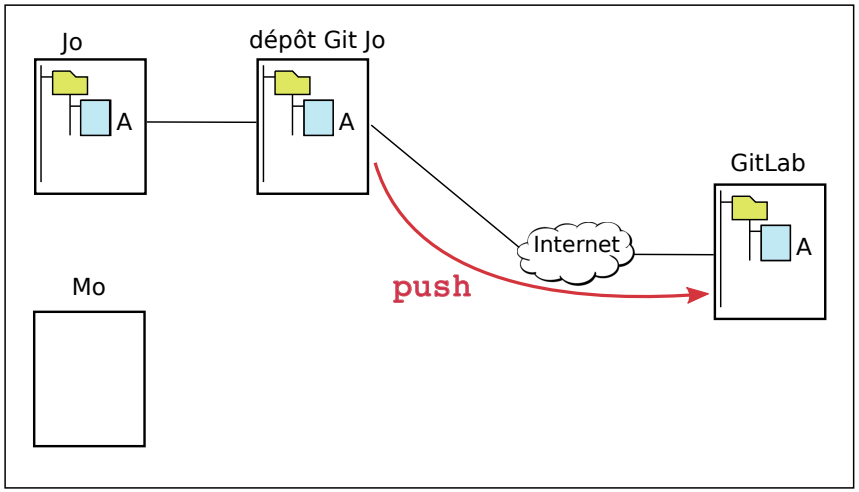
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo envoie les nouveaux fichiers/répertoires vers **son** dépôt Git



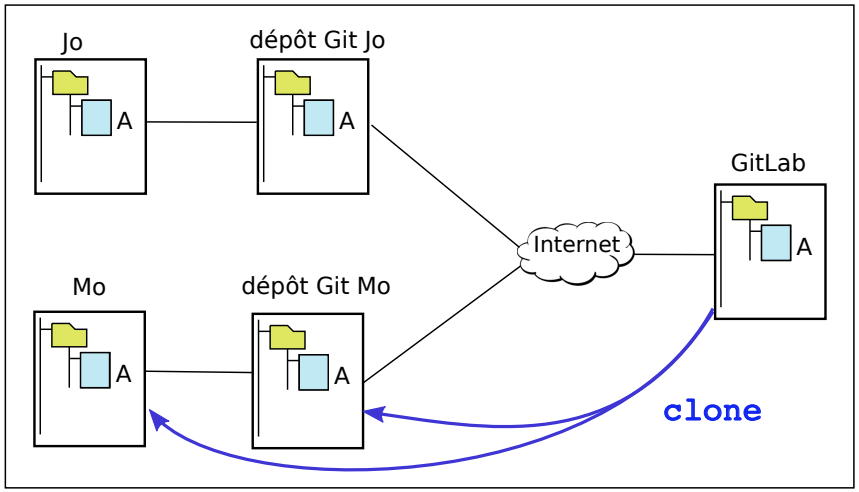
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo pousse ses modifications vers le GitLab



Jo et Mo avec Git (l'initialisation, une autre méthode)

Mo fait une copie (synchronisée) du dépôt sur le GitLab



Jo, Mo et Git, en vrai...

- Git s'utilise dans un terminal à l'aide de la commande `git`
- Il existe une multitude de clients graphiques pour Git...
- En TP nous utiliserons : Git en ligne de commande

Remarque 2 (Le GitLab ISTIC)

- `https://gitlab.istic.univ-rennes1.fr/`
- *Connectez-vous (au moins une fois) sur le GitLab istic avant le prochain TP !*
- *Créez un projet et tentez d'y déposer des fichiers*
- *Le code de vos projets GitLab sera consultables par un navigateur (Démo)*

Jo, Mo et Git, dans le terminal (initialisation)

Exemple 1 (Jo clone un projet gen1 auquel il a accès par le GitLab)

```
% git clone https://gitlab.istic.univ-rennes1.fr/jo/gen1.git
% cd gen1     # le répertoire de travail gen1 a été crée par git
```

Exemple 2 (Jo ajoute le fichier `lettre.txt` au projet (staging))

```
% git add lettre.txt
```

Exemple 3 (Jo envoie les modifications vers son dépôt Git **local)**

```
% git commit -m "ajout d'une lettre pour le père noël"
```

Exemple 4 (Jo pousse ses modifications vers le GitLab)

```
% git push
```

Exemple 5 (Mo clone le projet gen1 auquel il a accès par le GitLab)

```
% git clone https://gitlab.istic.univ-rennes1.fr/jo/gen1.git
% cd gen1
```


Jo, Mo et Git, dans le terminal (utilisation)

On suppose que le répertoire de travail est `gen1` (on a fait `cd gen1`)

Exemple 6 (Mo signale une modification sur `lettre.txt` (staging))

```
% git add lettre.txt
```

Exemple 7 (Mo envoie les modifications vers son dépôt Git **local)**

```
% git commit -m "ajout de remerciements"
```

Exemple 8 (Mo pousse ses modifications vers le GitLab)

```
% git push
```

Exemple 9 (Jo obtient les modifications faites sur le GitLab)

```
% git pull
```


Jo, Mo et Git, en vrai... dans le terminal (III)

Quiz 1 (Cette suite de commande échoue ? V Oui | R Non)

```
% git add monFichier
% git commit -m "modifications sur monFichier"
% git add monFichier
% git commit -m "on commite sans avoir modifié monFichier"
```

Quiz 2 (Cette suite de commande détruit définitivement `monFichier` ?)

```
% rm monFichier
% git commit -m "Suppression de monFichier"
% git push
```

V	*Oui*
R	*Non*

Quiz 3 (Cette suite de commande détruit définitivement `monFichier` ?)

```
% git rm monFichier
% git commit -m "Suppression de monFichier"
% git push
```

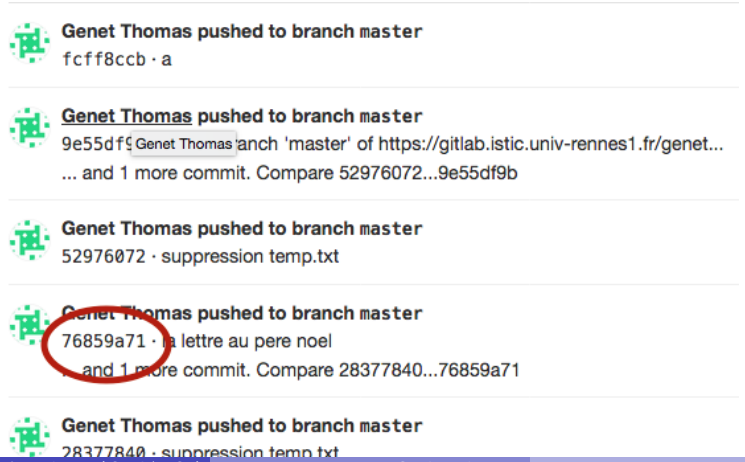
V	*Oui*
R	*Non*

Dans "gestionnaire de version", il y a le mot "version"

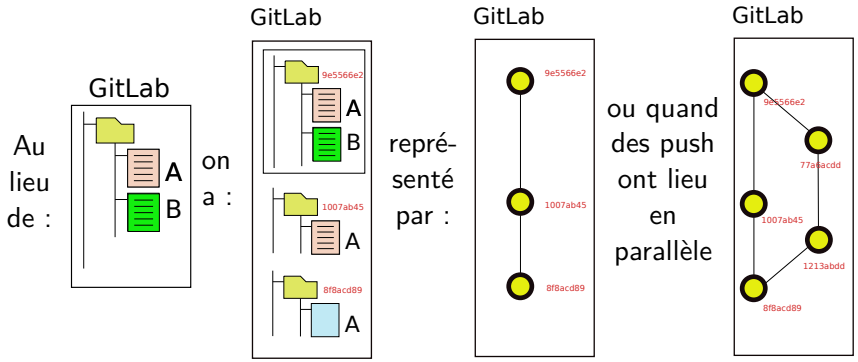
Remarque 3 (En Git chaque version a un identifiant unique)

Git conserve l'historique complet de toutes les versions du projet.

Dans GitLab, l'onglet "Activity" donne les identifiants des versions



Dans "gestionnaire de version", il y a le mot "version"



Dans "gestionnaire de version", il y a le mot "version"

Exemple 10 (Jo replace tout son projet dans la version 76859a71)

```
% git reset --hard 76859a71
```

Attention ! on perd les modifications du projet non "commitées" dans Git !

Exemple 11 (Jo restaure le fichier `temp1.txt` de la version courante)

```
% git checkout temp1.txt
```

Attention ! on perd les modifications sur `temp1.txt` non "commitées" dans Git !

Dans "gestionnaire de version", il y a le mot "version"

Exercice 1

1. *Comment récupérer un fichier détruit dans le répertoire local ?*
2. *Comment récupérer un fichier supprimé de la dernière version du GitLab ?*

Quiz 4

Initialement, on suppose que le projet est dans la version 52976889 *et que l'on tape :*

```
% rm temp1.txt
% git reset --hard 76859a71
```

Comment remettre le projet dans sa forme initiale ?

`% git reset --hard 52976889`	`% git checkout temp1.txt`
V	*R*

La vie n'est pas un long fleuve tranquille : les conflits

Définition 12 (Conflit de version)

Si deux développeurs font des modifications contradictoires sur un même fichier, un conflit peut apparaître. Il touchera le dernier des deux faisant un `push`. Ce dernier sera le seul à pouvoir résoudre le conflit.

En pratique, si vous suivez les principes de **modularité** présentés aux cours précédents, les conflits de versions devraient être très rares, mais... (voir le titre)

Exemple 13 (Exemple de conflit de version)

Jo et Mo modifient différemment la ligne *i* du fichier B de la dernière version. Jo fait son `push`. Ensuite, Mo tente de faire son `push`. Celui-ci échoue car sa version n'est plus à jour. Mo fait un `pull` qui révèle un conflit sur la ligne *i* du fichier B.

Résolution de conflit Git sur un fichier B

1. Rechercher les annotations de conflits dans le texte du fichier B :

```
Le texte précédant la partie de texte en conflit.
<<<<<<< HEAD
C'est le texte que je voulais proposer.
=======
C'est le texte proposé par l'autre développeur.
>>>>>>> e4e1e956f6426c04ab80487f612265fe5c95ece8
La suite du texte qui n'est pas en conflit.
```

2. Résoudre le conflit dans le texte du fichier B :

```
Le texte précédant la partie de texte en conflit.
C'est le texte que je voulais proposer complété par le
texte proposé par l'autre développeur.
La suite du texte qui n'est pas en conflit.
```

3. Informer le dépôt de la résolution du conflit sur B :

```
% git add B
% git commit -m "Conflit sur 2eme ligne résolu"
% git push
```


Résolution de conflit Git sur un fichier B (II)

Quiz 5 (Ces séquences de commandes sont-elles possible ?)

Sur la machine de Jo	*Sur la machine de Mo*
`% cat temp1.txt`	`% cat temp1.txt`
`Le texte de ce document`	`Le texte de ce fichier`
`est très court`	`est vraiment très court`
`% git add temp1.txt`	`% git add temp1.txt`
`% git commit -m "modif Jo"`	`% git commit -m "modif Mo"`
`% git push`	`% git push`
`% git pull`	`% git pull`

Si Mo fait son push avant Jo, il aura un conflit — V Vrai | R Faux

V Oui | R Non