Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2020 – 2021
DEV1 – ENVL

DEV1 – ENVL – Laboratoire d'environnement système

TD 5 – Git - Console

Un logiciel de gestion de versions est un outil permettant de conserver l'historique des modifications apportées à un projet (code source, documentation, etc.).

L'entièreté de l'historique étant conservé, les fichiers peuvent être comparés avec n'importe quelle version précédente. Un tel logiciel permet également de maintenir les modifications d'un code source cohérentes à travers le temps, en particulier si on travaille en équipe. Lorsque deux développeurs modifient un même fichier, ils en seront notifié pour éviter que l'un écrase le travail de l'autre. Ils pourront alors fusionner leurs codes ensemble. L'utilisation d'un logiciel de gestion de versions est devenue essentielle dans une équipe et est très appréciable lorsque l'on travaille seul.

Il existe une multitude de logiciels de gestion de versions, dont l'un s'est démarqué au cours de la dernière décennie : **git**.

Préalable

Nous supposons dans ce TD que le TD « Git - Netbeans » a été réalisé.

Table des matières

Version du document
12 septembre 2020

1 Vocabulaire

Pour permettre l'accès à l'historique d'un projet, **git** doit conserver toutes ces données de manière structurée. L'espace de stockage contenant toutes ces données est nommé **dépôt** (*repository*).

Les éléments principaux composant un dépôt sont les **soumissions** (*commits*). Chaque *commit* contient l'ensemble des fichiers et leurs contenus qui composaient le projet à un moment donné. Un commit contient également une série de méta-données comme une date de création, le nom du créateur et une description. Pour pouvoir les identifier, les *commits* possèdent tous une clé unique.

Pour partager facilement un dépôt entre plusieurs personnes, celui-ci doit leur être accessible. Pour faciliter cette gestion et l'hébergement d'un dépôt, plusieurs services web ont vu le jour pour conserver un dépôt centralisé. Citons *Github*, *GitLab*, et *Bitbucket*.

1.1 Exemple

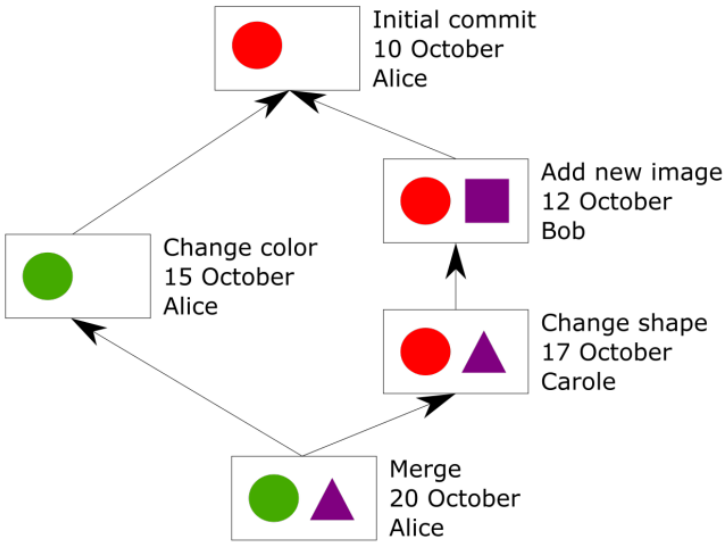


FIGURE 1 – Exemple d'évolution d'un projet.

La figure 1 illustre l'évolution possible d'un projet.

1. Alice crée d'abord un premier *commit* avec un fichier simple : l'image d'un cercle rouge.
2. Ensuite, Bob ajoute un nouveau *commit* avec l'image d'Alice et un nouveau fichier contenant un carré mauve.
3. Carole modifie le carré mauve de Bob en triangle.
4. Pendant ce temps, Alice, dans un nouveau commit, change la couleur de son cercle, sans être au courant des changements de Bob et Carole.
5. Enfin, les deux équipes se rendent compte de leur dispersion et créent un nouveau *commit* avec les améliorations de chacun.

Commande git

Une commande **git** est de la forme :

```
git <command> [options]
```

À chaque commande (`command`) correspond des options éventuelles (`options`).

2 Mise en place

Lorsque plusieurs personnes travaillent ensemble, elles peuvent centraliser leurs dépôts sur internet. Par exemple sur un serveur *Gitlab*.

Ces personnes peuvent donner la possibilité à d'autres de copier tout leur dépôt. Cette copie sera un dépôt complètement indépendant du premier. On parle de *fork*.

Nous travaillons avec le dépôt `https://git.esi-bru.be/dev1/labo-envl-git` qui est en lecture seule et que nous vous invitons à *forker*.

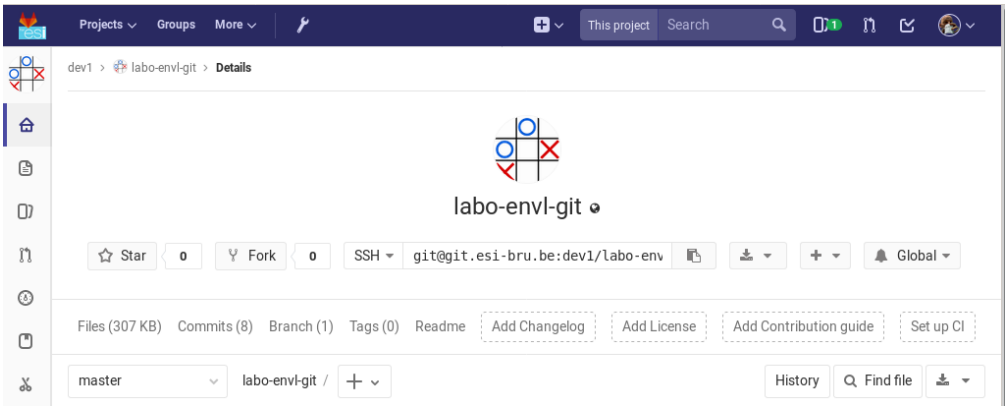
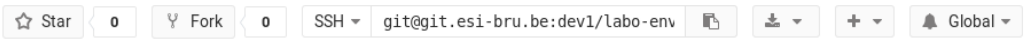


FIGURE 2 – Dépôt git à l'adresse `https://git.esi-bru.be/dev1/labo-envl-git`

Exercice 1 Forker le dépôt

- Se rendre à l'adresse `https://git.esi-bru.be/dev1/labo-envl-git`. Forkez le projet en cliquant sur le bouton idoine.



- Prenez note de l'adresse du *fork* qui sera de la forme : `https://git.esi-bru.be/<votrelogin>/labo-envl-git`.
  À cette adresse, vous trouverez l'adresse de votre dépôt dans deux protocoles différents : `ssh` et `https`. Voir les images à la figure 3 page suivante.

Pour travailler localement sur un dépôt, il est nécessaire de le **cloner** c'est-à-dire rapatrier son contenu sur la machine pour y avoir accès. C'est la commande `clone` de git qui s'en charge.

FIGURE 3 – Un dépôt git a deux liens : https et ssh

Cette commande crée un répertoire et copie le dépôt. Dans le répertoire copié se trouve un sous-répertoire `.git` contenant tout ce qui concerne git.

Exercice 2 Cloner le projet localement

- Clonez le dépôt `labo-envl-git` dans le répertoire courant en utilisant l'adresse de votre fork commençant par `https`.

```
$
git clone https://git.esi-bru.be/login[a]/labo-envl-git
```

a. Remplacer 'login' par votre login

terminal

Git vous demande vos identifiants. Entrez les.

- Constatez la présence d'un nouveau répertoire `labo-envl-git` dont le contenu à l'allure de la figure 4

```
pbt@foxtrot:~> tree labo-envl-git/ -a -L 1
labo-envl-git/
├── .git
├── .gitignore
├── logo.png
├── main.java
├── readme.md
└── test.java
```

FIGURE 4 – Contenu du répertoire après clone

Par défaut **git** utilise vos nom et email tels que configurés par défaut sur la machine. Ceux-ci seront visibles par toutes les personnes ayant accès au dépôt. Il est donc important que ces noms reflètent correctement votre identité. Pour ce faire, il faut les enregistrer une fois correctement.

Exercice 3 Bien se présenter

- Enregistrez vos prénom et nom dans la configuration de git. Par exemple si vous vous appelez Juste Leblanc :

```
$
git config --global user.name "Juste Leblanc"
```
terminal

Enregistrez votre adresse email dans la configuration de git. Par exemple, si vous vous appelez Juste Leblanc et êtes étudiant de la he2b :

```
$
git config --global user.email jleblanc@etu.he2b.be
```
terminal

3 Historique

La commande **log** de git permet de voir l'historique des *commits*. En voici quelques options.

```
git log

    -n nombre
        Affiche les 'nombre' dernier commits.

    --oneline
        Historique sur une ligne avec un identifiant court.

    --name-status
        Affiche pour chaque commit la liste des fichiers
        et leur état.

        Quelques états sont (en anglais) : Added (A), Copied (C),
        Deleted (D), Modified (M), Renamed (R), have their
        type (i.e. regular file, symlink, submodule...) changed (T),
        are Unmerged (U)...
```

Chaque *commit* est présenté chronologiquement, du dernier au premier, avec son identifiant, son auteur, sa date de création et son message. Par exemple le premier *commit* est :

```
commit 8da44fddfe53771a46202a1d9512496b20609dd6
 Author: denis <denisname@users.noreply.github.com>
 Date:   Thu Oct 17 13:02:21 2019 +0200

     commit initial

     Contient le lisez-moi, le logo et un "main".
     Mais  pas encore de code.
```

Exercice 4 Observer l'historique des commits

- Comptez le nombre de *commits* de votre dépôt. Dans quelle période de temps (date et heure) les *commits* ont-ils étés créés ?
- Comparez les identifiants complets aux identifiants courts.

L'option `--name-status` affiche, pour chaque *commit* la liste des fichier et leur statut. Par exemple, si pour un *commit* on a :

```
9f35c98 Une petite description
    A logo.png
    D logo.jpg
    M code.c
    R100 avant.txt apres.txt
```

Ceci veut dire :

1. le fichier `logo.png` a été ajouté ;
2. le fichier `logo.jpg` a été supprimé ;
3. le code C se trouvant dans le fichier `code.c` a été modifié ;
4. le fichier `avant.txt` a été renommé en `apres.txt`. Les deux fichiers sont 100% identiques. Ils n'ont pas subi de modification. (git détecte parfois un renommage alors que vous avez simplement supprimé un fichier et créé un autre identique ou très similaire. Cela n'a aucune importance pour git.)

Exercice 5 Statuts des *commits*

- À quel moment le fichier `test.java` a été ajouté ?
- Lors de quel commit, le fichier `lisezmoi.md` a été renommé en `readme.md` ?
- Durant quel commit, le fichier `todo.txt` a-t-il été supprimé ?
- Quel est le seul *commit* où `logo.png` a été modifié ?

Exercice 6 Affichage

- Affichez les changements des deux derniers *commits* uniquement.

4 Comparaison de *commits*

La commande **`diff`** de git permet de vérifier quelles modifications ont été apportées aux fichiers.

```
git diff <id1> <id2>

Compare le commit id1 avec le commit id2
    id1 et id2 sont les premiers caractères identifiant un commit
```

Exercice 7 Utilisation de diff

- Comparez le contenu des fichiers entre le premier et le dernier commit.
- Comparez le contenu du fichier `todo.txt` avant et après sa suppression.
- Comparez le contenu de `readme.md` avant et après avoir été renommé.
- Comparez le contenu de `logo.png` avant et après modification.

Le dernier *commit*, en plus de son identifiant, a comme nom symbolique **`HEAD`**. Les *commits* précédents peuvent être identifiés par : `HEAD~1`, `HEAD~2`, `HEAD~3`, etc.

Exercice 8

- Comparez le contenu des fichiers entre les deux derniers *commits*, sans utiliser leurs identifiants.
- Notez la différence entre les deux commandes :
  `git diff id1 id2` et `git diff id2 id1`.

5 État du dépôt local et zones

La commande **`status`** de git donne l'état du dépôt. La commande **`add`** de git ajoute des fichiers à la zone de transit. Les commandes **`mv`** et **`rm`** de git déplacent et suppriment un fichier.

Ces deux commandes agissent par défaut sur la zone de transit en plus de la zone de travail (voir encadré ci-après), contrairement aux commandes `mv` et `rm` que vous connaissez déjà, qui elles n'agissent que sur la zone de travail.

```
git status

git add <files>
git mv <file1> <file2>
git rm <file>
```

Si aucun changement n'a été fait, un `git status` nous informe que tout va bien comme ceci :

```
On branch master
nothing to commit, working directory clean
```

Par exemple, après une modification du fichier `fichier.ext`, un `git status` nous donnerait :

```
On branch master
Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)

modified: fichier.ext

no changes added to commit (use "git add" or "git commit -a")
```

Git détecte que le fichier a été modifié mais avertit que les changements n'ont pas été déplacés en zone de transit avant *commit* (*Changes not staged for commit*).

Zone de travail - zone de transit - dépôt

Il existe trois zones dans un dépôt git :

- la **zone de travail** (*working directory*) est celle du disque dur, du répertoire de travail, celle dans laquelle nous effectuons les modifications ;
- la **zone de transit** (*staging area* ou encore *index*) est une zone entre la zone de travail et le dépôt dans laquelle on « charge » les changements de la zone de travail qui seront envoyés au prochain *commit*.

  Cette zone permet de construire ses *commits* indépendamment des modifications effectuées. Par exemple, lors de l'ajout d'une nouvelle fonctionnalité, il est possible de faire toutes les modifications de code jusqu'à ce que la tâche soit effectuée et commiter ensuite les changements par petits groupes d'ajouts plus simples à comprendre par la suite.
- le **dépôt** (*git directory* ou *repository*) proprement dit contenant tous les *commits*.

Attention que le *dépôt local* dont nous parlons maintenant n'est peut-être pas synchronisé avec le *dépôt distant*.

Exercice 9 Modification de dépôt

- Certaines lignes du fichier `main.java` sont trop longues. Coupez les lignes 31, 37, 42 et 46 aux bons endroits. Sauvez votre fichier et observez le résultat produit par un `git status`.

  Les modifications ne sont pas validées. C'est normal.
- Ajoutez les changements à la zone de transit avec `git add main.java` et observez l'état de votre dépôt avec `git status`.

  Les modifications sont maintenant validées et prêtes à être *commitées* (voir figure 5). Elles sont passées de la zone de travail (*working directory*) à la zone de transit (*staging area*).

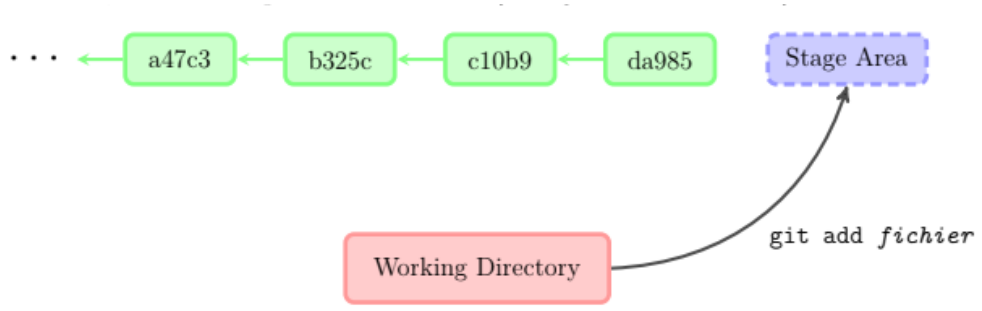


FIGURE 5 – État du dépôt lors de l'ajout d'un fichier à la zone de transit

Exercice 10 Renommer un fichier (version longue)

- La classe s'appelle `main`, ce qui n'est pas un bon choix de nom. Renommez la classe en `Main` (sans renommer le fichier pour l'instant).

▷ Observez le résultat de `git status`.

▷ Faites un nouveau `git add main.java` et observez le résultat.

✎ Si la classe s'appelle `Main`, le fichier doit s'appeler `Main.java`. Pour le renommer, vous entrez naturellement la commande `mv main.java Main.java`. Faites le et observez le résultat avec un `git status`.

Git vous signale que le fichier `main.java` est supprimé et qu'un nouveau fichier `Main.java` — non suivi — existe.

✎ Pour ajouter le nouveau fichier, vous entrez naturellement la commande `git add Main.java`. Faites le et observez le résultat.

Git vous dit alors qu'un nouveau fichier `Main.java` existe, que le fichier `main.java` est modifié et que sa suppression n'est pas validée !

✎ Vous entrez alors la commande `git add main.java` pour valider la suppression. Faites le et observez le résultat.

Git vous montre (enfin) que le fichier a été renommé. Nous avions dit : « Version longue » n'est-ce-pas ?

Exercice 11 **Renommer un fichier (version courte)**

Finalement, cette classe pourrait s'appeler `Joxo` plutôt que `Main`.

✎ Éditez le fichier et renommez la classe.

✎ Renommez le fichier d'un simple `git mv Main.java Joxo.java` et vérifiez le résultat avec un `git status`... et un `ls`.

6 Comparaison de *commits* et de zones

Nous avons déjà vu comment comparer des *commits* entre eux. Voyons comment comparer la zone de travail, la zone de transit et des *commits*. La figure 6 page suivante facilitera votre compréhension.

```
git diff
    Compare la zone de transit avec la zone de travail
git diff --staged
    Compare le dernier commit avec la zone de transit
git diff <id>
    Compare le commit id avec la zone de travail
git diff --staged <id>
    Compare le commit id avec la zone de transit
git diff <id1> <id2>
    Compare le commit id1 avec le commit id2
```

Exercice 12 **Comparaison zone de travail et zone de transit**

✎ Éditez le fichier `Joxo.java` et ajoutez un commentaire javadoc pour la classe.

✎ Comparez la zone de travail et la zone de transit avec `git diff`.

Git vous montre l'ajout du commentaire javadoc.

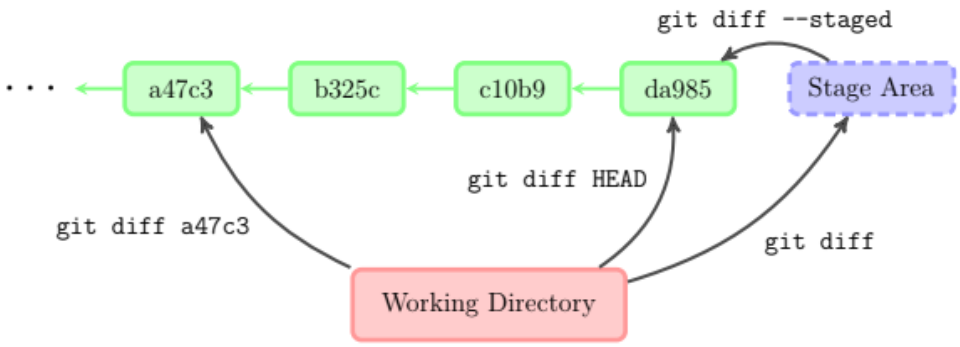


FIGURE 6 – Différentes commandes de comparaisons

Exercice 13 **Comparaison zone de travail et dernier commit**

Comparez la zone de travail et le dernier *commit* avec l'une des deux méthodes suivantes :

- `git diff HEAD` puisque le dernier *commit* s'appelle `HEAD` ;
- `git log --oneline` pour trouver le numéro du dernier commit, suivi de `git diff bca37`.

Git vous montre l'ajout du commentaire javadoc ainsi que les modifications apportées aux lignes trop longues.

Exercice 14 **Comparaison zone de transit et dernier commit**

Comparez la zone de transit et le dernier *commit* en ajoutant `--staged` aux dernières commandes.
Git ne vous montre plus l'ajout du commentaire javadoc qui est la modification dans la zone de travail mais uniquement les modifications apportées aux lignes trop longues du commit `bca37`.

7 Création de *commits*

```
git commit
    Commite les changements en ouvrant l'éditeur par défaut.

    -m <message>
        Commite directement les changements en utilisant le message
        passé en paramètre comme message de commit.
```

Une fois les fichiers ajoutés à la zone de transit et les dernières vérifications faites, nous pouvons enfin *commiter* les fichiers pour les placer dans le dépôt local.

L'opération terminée, git affiche un message contenant, entre autres, l'identifiant court du *commit*, la première ligne de message et le nombre de changements.

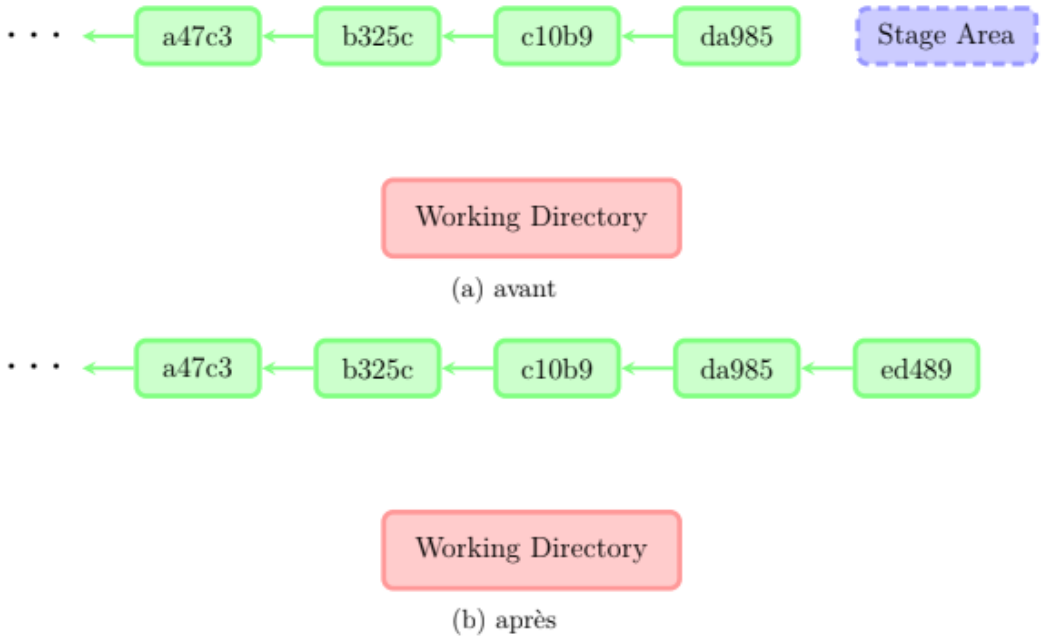


FIGURE 7 – Situation d'un *commit* « avant / après »

Par exemple :

```
[master ed489ba] First commit
1 file changed, 2 instertions(+)
```

Si vous exécutez la commande `git commit` sans l'option `-m`, git ouvre l'éditeur de texte par défaut du système [1] pour permettre l'écriture du message. Il est ainsi possible d'écrire un message avec une explication sur plusieurs lignes.

Il n'y a pas de contrainte sur la forme des messages. Néanmoins, la convention la plus utilisée est de commencer le message par une courte ligne de moins d'une cinquantaine de caractères, suivie d'une ligne vide puis d'une description complète. La première ligne servira de titre, par exemple, dans un `git log --oneline`.

Exercice 15 **Commit de nos changements**

Notre dépôt contient un changement dans la zone de transit et un changement dans la zone de travail. Nous allons faire deux commits.

- Dans la zone de transit se trouve le changement concernant la longueur des lignes. Faites un *commit* avec `git commit -m "Correction de la longueur des lignes"`
  Vérifiez que le *commit* est bien fait avec un `git log` par exemple.
- Occupons nous du *commit* concernant la javadoc dont les modifications se trouvent uniquement dans la zone de travail. Que faut-il faire ?
  - ▷ Ajoutez les changements dans la zone de transit avec `git add Joxo.java`
  - ▷ Commitez avec un `git commit`.
    Git ouvre l'éditeur par défaut. Écrivez un message court suivi d'une message long présentant votre commit.

1. Sur de nombreux systèmes, cet éditeur de texte par défaut est `vim`. Son utilisation n'est pas intuitive. Si vous préférez nano, la commande `git config --global core.editor nano` demandera à git d'enregistrer votre choix.

8 Publication des changements

```
git push
    Pousse à partir du dépôt local vers le dépôt distant.
git pull
    Tire à partir du dépôt distant vers le dépôt local.
```

Après les *commits* de la section précédente, le dépôt devrait avoir cette allure :

```
commit 210eb38e9648bcabcd5bf9f59bfec1cffd714de1 (HEAD -> master)
Author: Pierre Bettens (pbt) <pbettens@he2b.be>
Date:   Fri Nov 15 11:27:36 2019 +0100

    ajout de la javadoc

    Ajout d'une javadoc et d'un commentaire de commit un peu plus long.

commit 77e542da5b02de89f6898a9f0ccafee9a7090821
Author: Pierre Bettens (pbt) <pbettens@he2b.be>
Date:   Fri Nov 15 11:25:26 2019 +0100

    Correction de la longueur des lignes

commit bca37893fc1e29e6dd0a7b0bc292c6cf8aba624c (origin/master, origin/HEAD)
Author: denis <denisname@users.noreply.github.com>
Date:   Thu Oct 17 15:25:30 2019 +0200

    Fixe test si le joueur gagne en oblique

    fixes #3
```

La première ligne, montre que le *commit* `210eb` est le dernier commit : `HEAD`. Le *commit* `bca37` est étiqueté **origin/master** et est le dernier *commit* connu du dépôt distant.

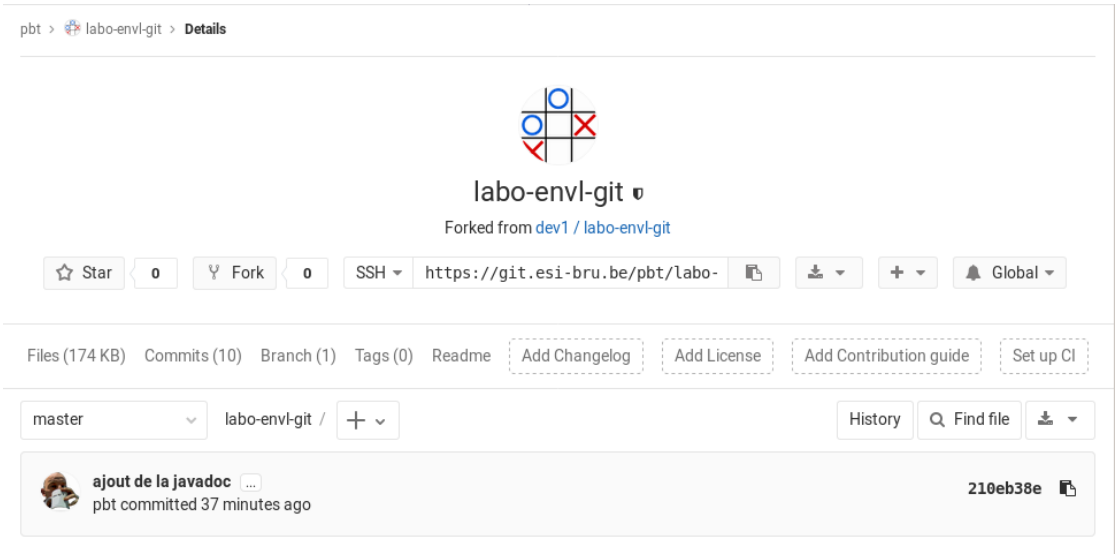
Pour synchroniser le dépôt local avec le dépôt distant, il faut téléverser (*uploader*) le dépôt local (ses commits) vers le dépôt distant.

Exercice 16 Mettre à jour le dépôt distant

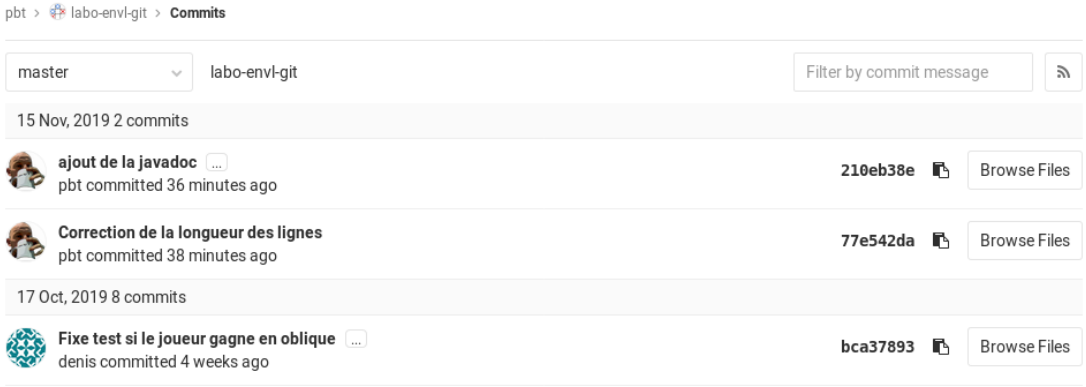
Faites un `git push` et vérifiez ensuite que les changements ont bien été répercutés sur le dépôt distant (à l'adresse que vous aviez notée). Voir figure 8 page suivante.

Rappel

Lors d'un travail à plusieurs ou sur des machines différentes, il est **indispensable** de s'assurer que le dépôt local est à jour avec le dépôt distant en exécutant un `git pull` **avant** de commencer à travailler.



Page du dépôt montrant le dernier commit.



Extrait de la liste des commits.

FIGURE 8 – Situation du dépôt distant après le `push`

FAQ

J'ai du entrer mes identifiants lors du `clone` et lors du `push`. Faut-il faire ça à chaque fois ?

Oui et il faudra le faire aussi lors l'un `pull`.
Pour ne plus le faire, il est possible — et conseillé — de « déposer une clé ssh sur le serveur ». Voir sur le serveur gitlab de l'ESI *Profile / Settings / SSH Keys*. Les informations sur comment faire se trouve en cliquant sur *Generate it*

Je suis allé voir le répertoire `.git` comme conseillé. J'ai également vu un fichier `.gitignore` qu'est-ce que c'est ?

Ce fichier contient des règles demandant à git d'ignorer certains fichiers du dépôt local.