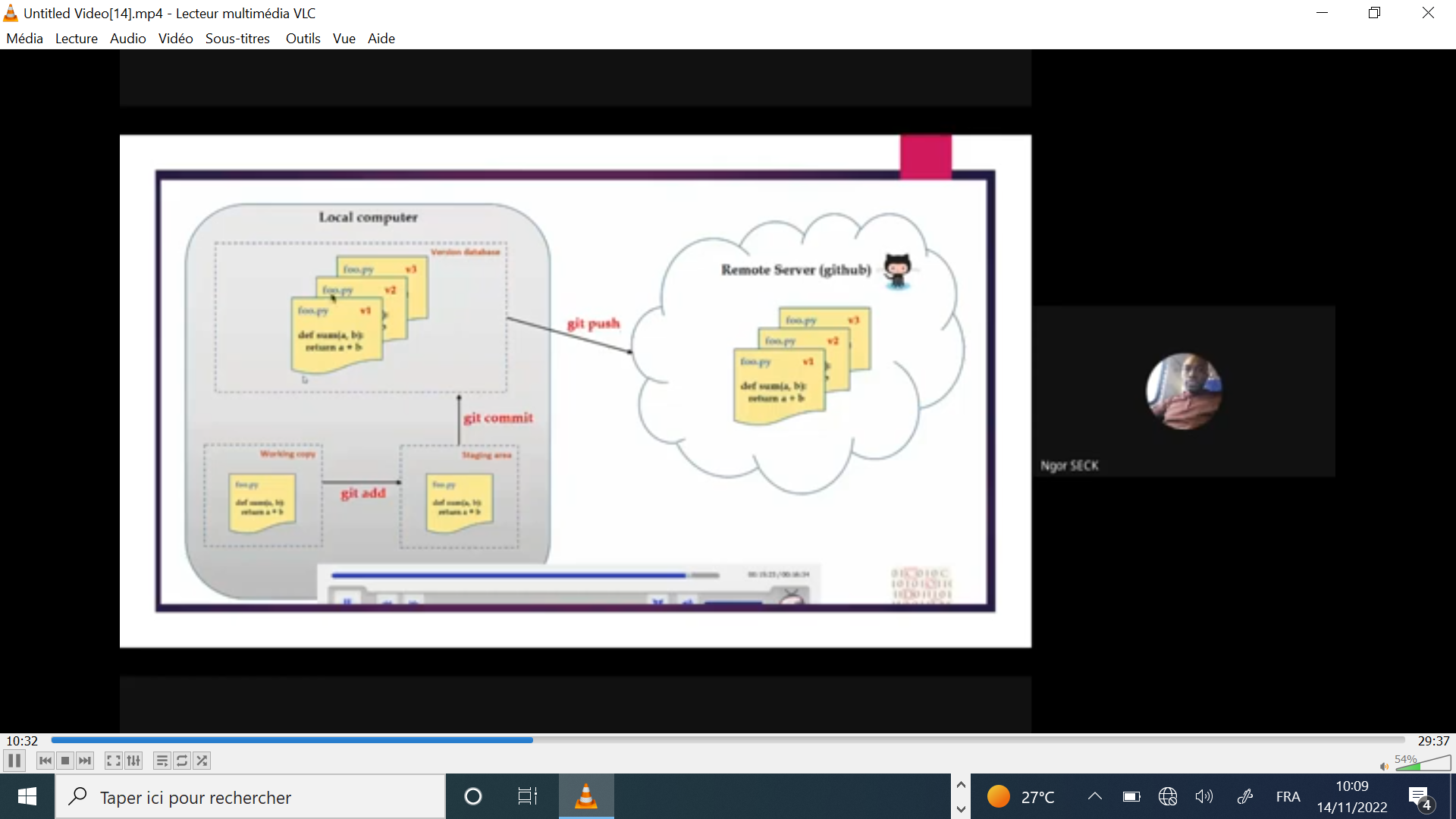
**GITHUB**



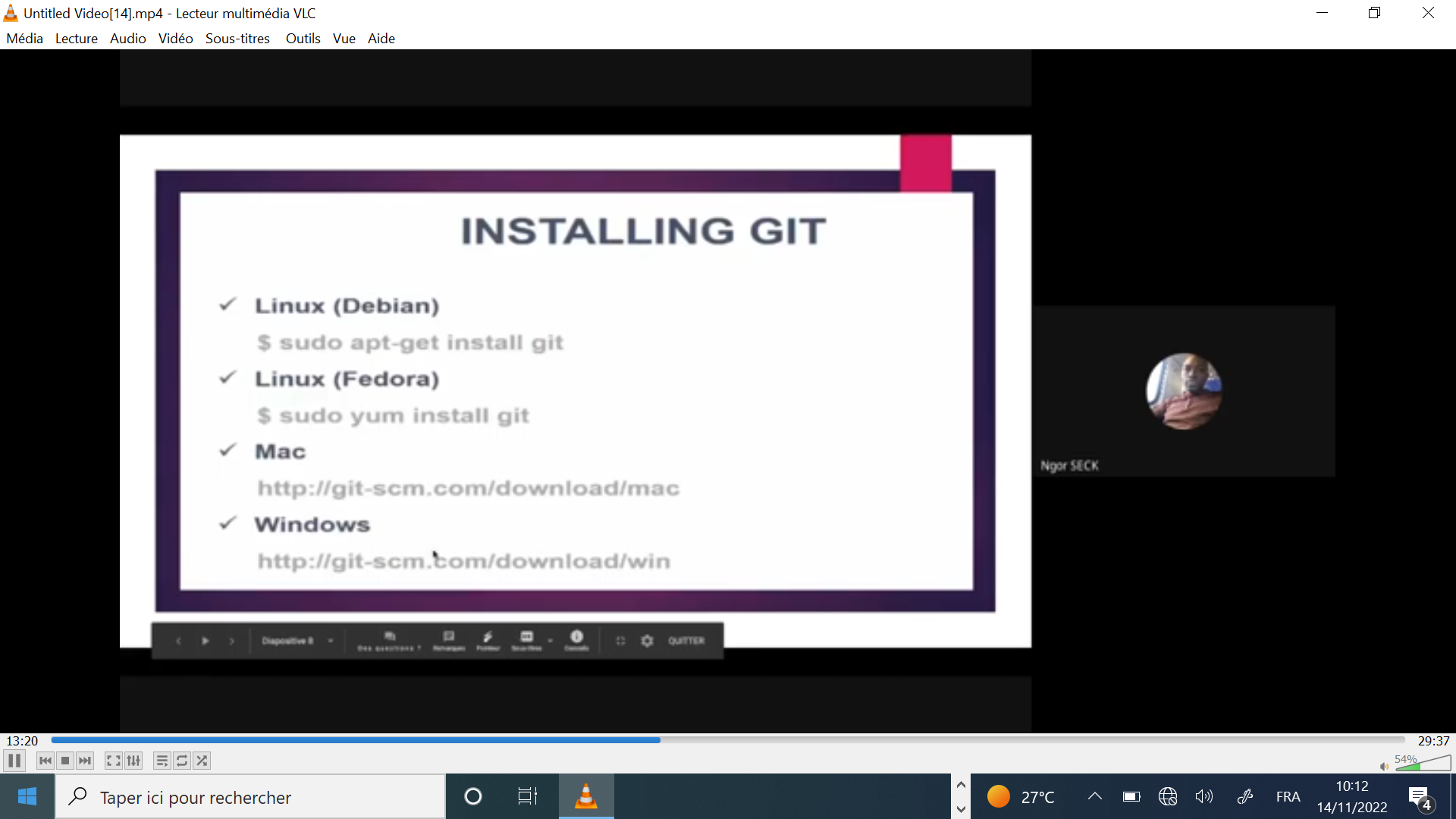
**git-push - Mettre à jour les références distantes avec les objets associés**

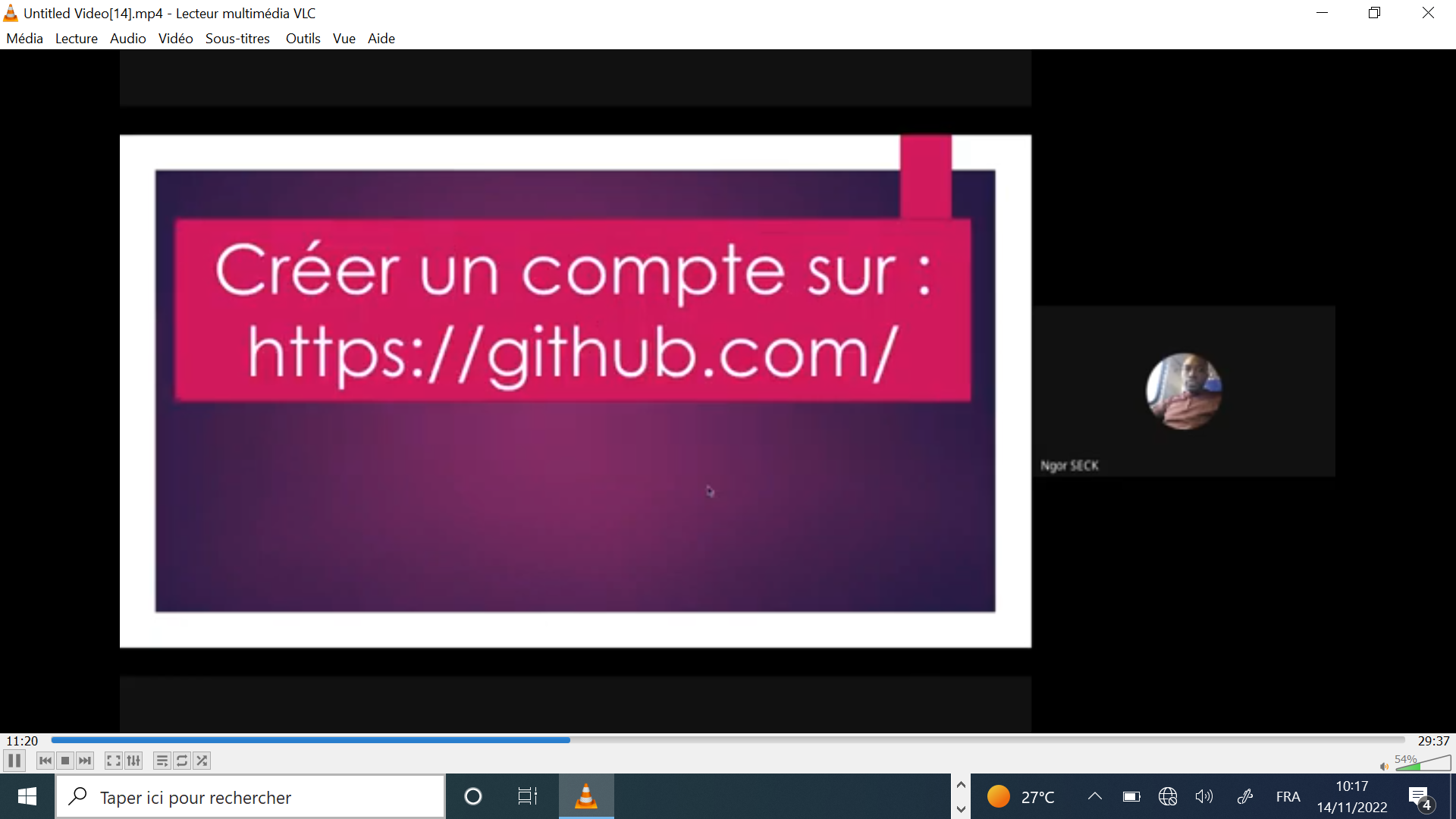
**git-remote - Gérer un ensemble de référentiels suivis**

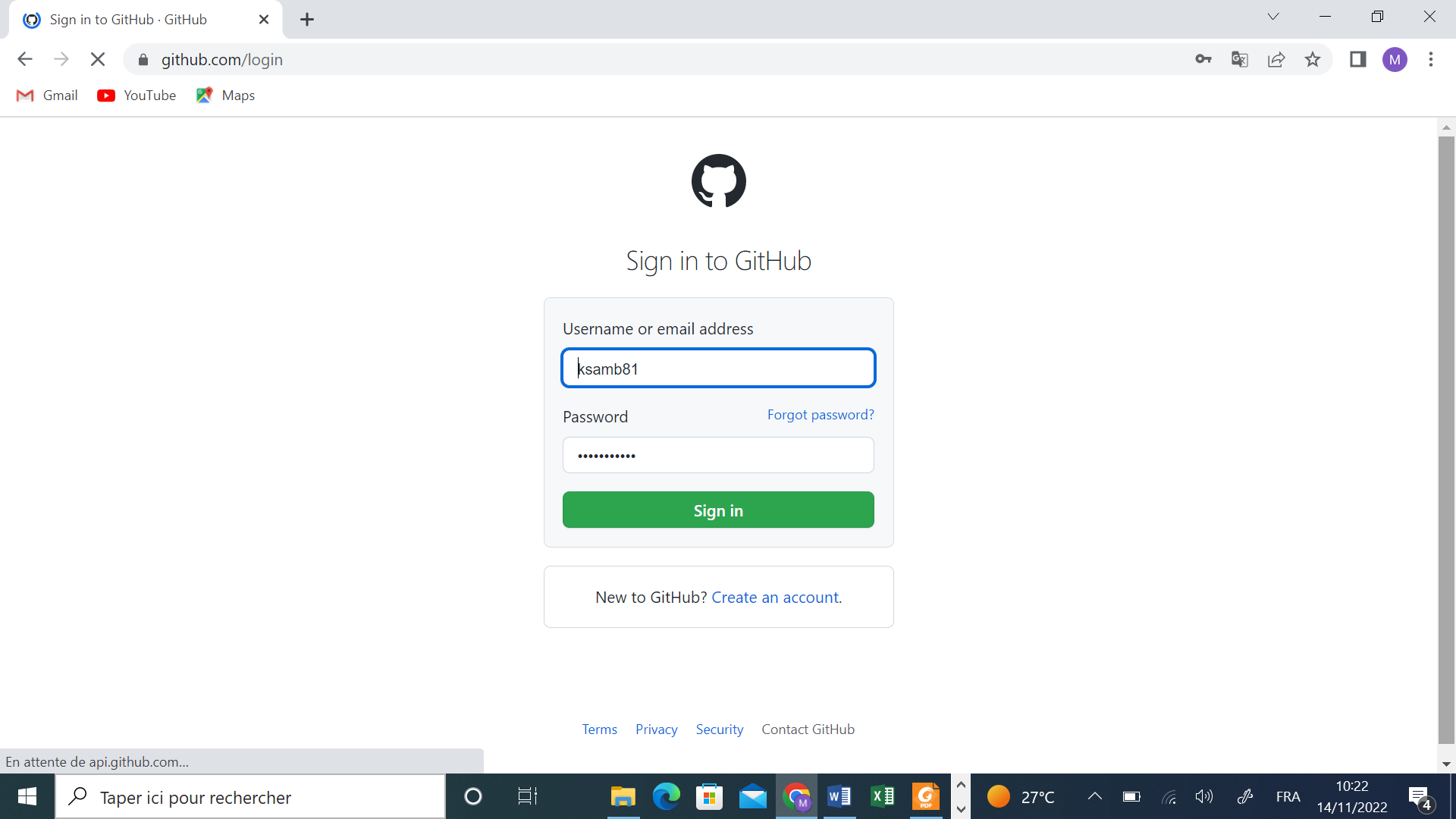
**git-commit - Enregistre les modifications apportées au référentiel**

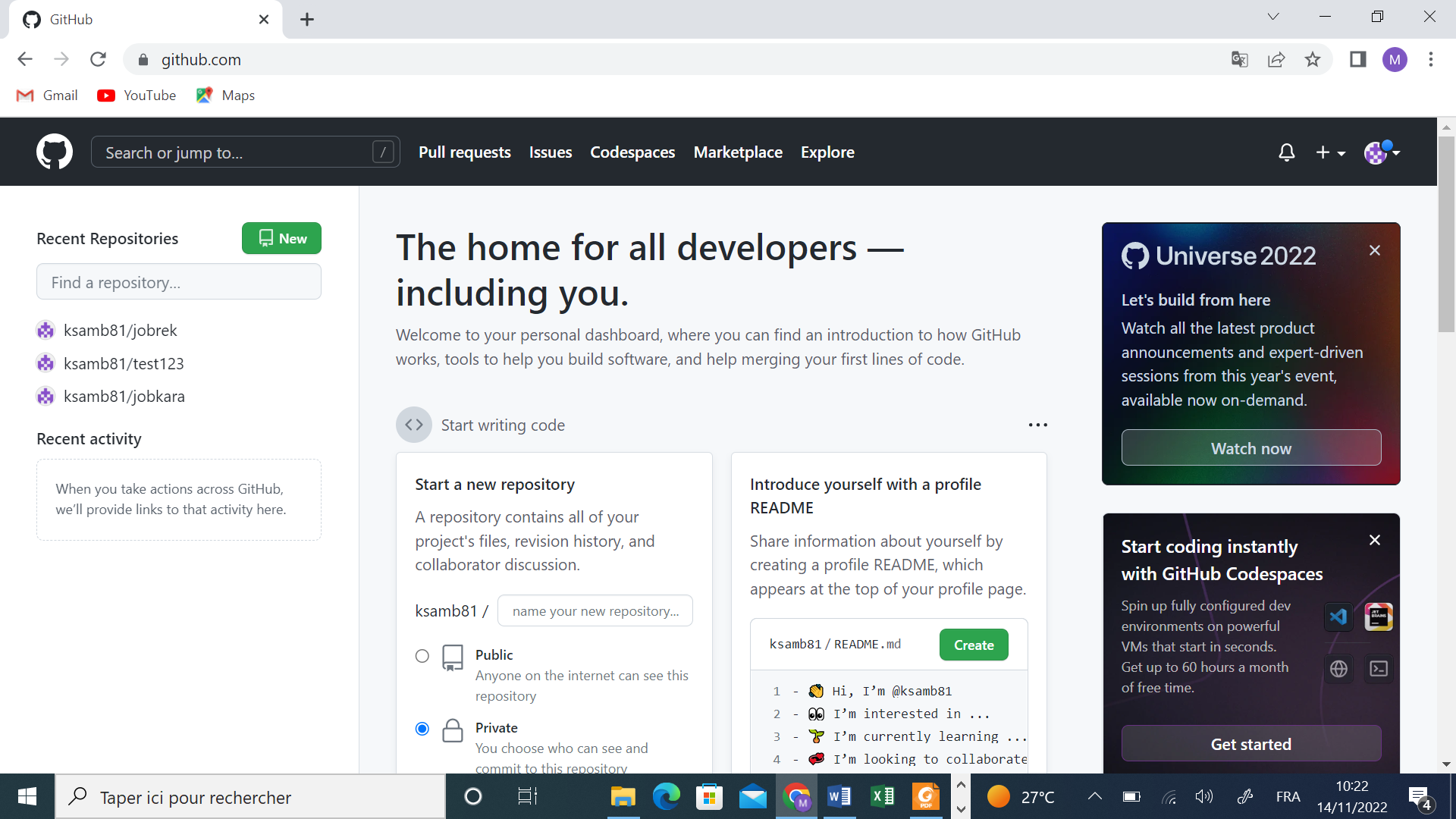
**git-add - Ajouter le contenu du fichier à l'index**

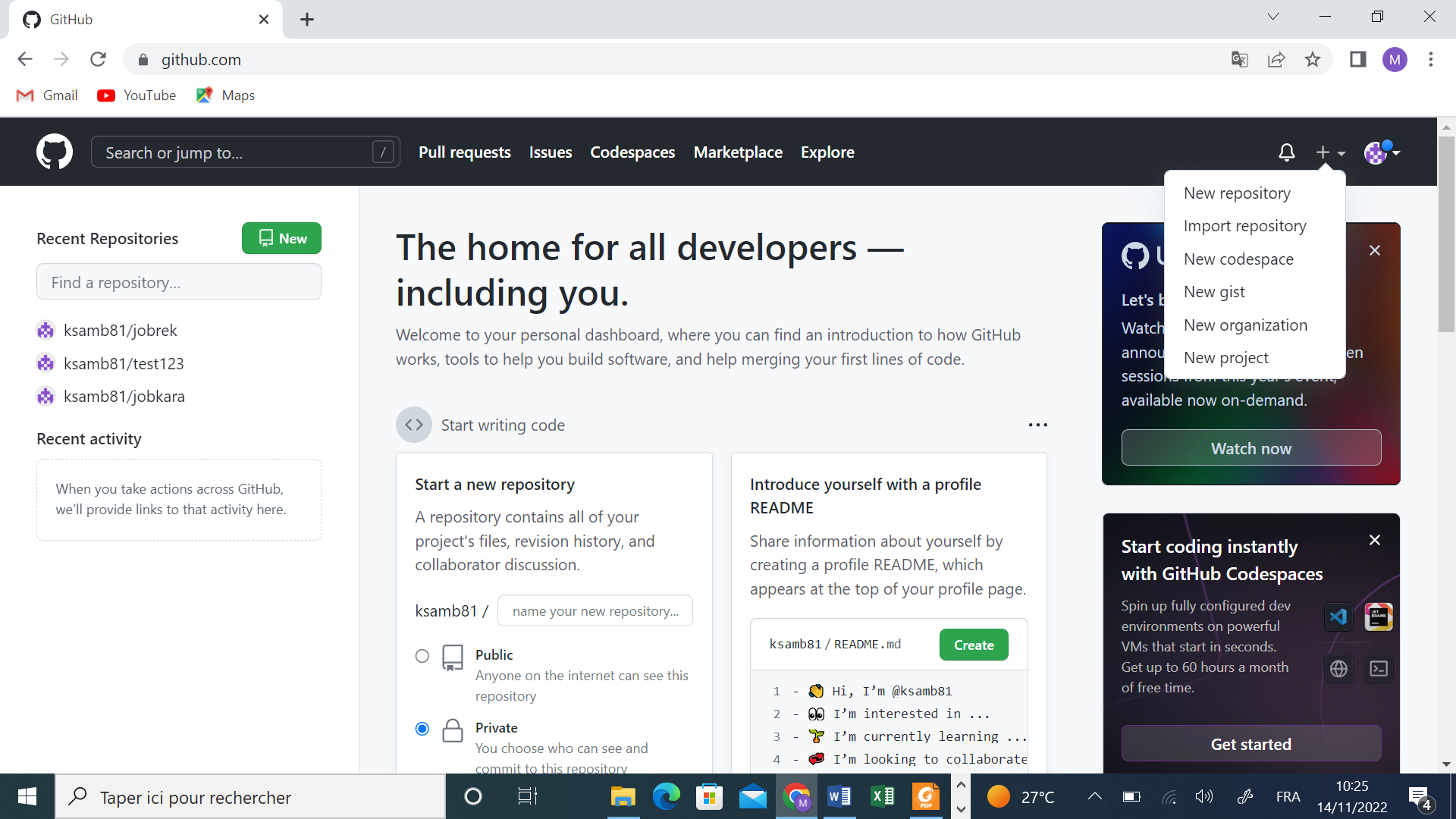
**Source**: https://git-scm.com/



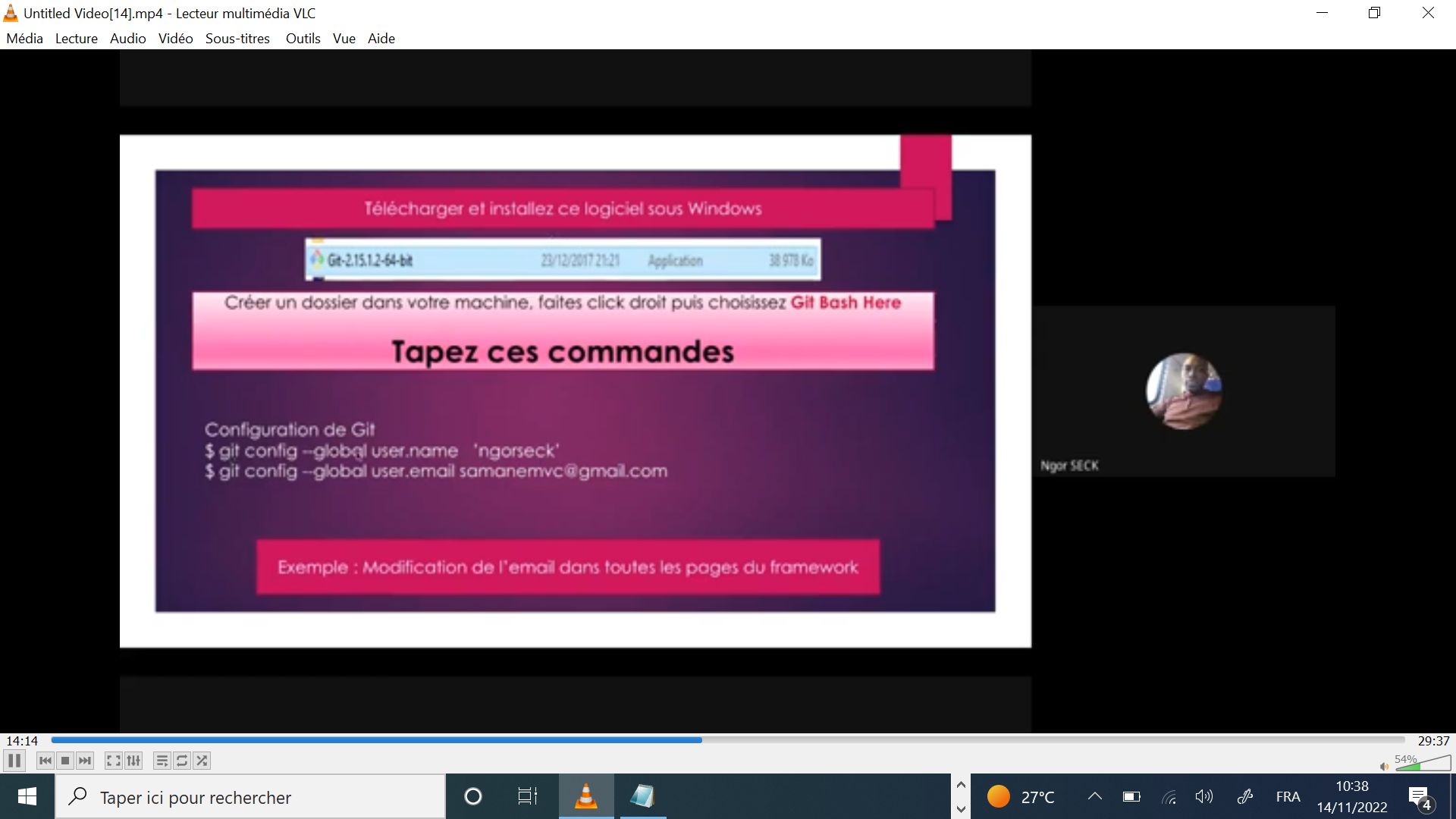








**CONFIGURATION DE GIT**



**git config --global –list**

**git config --global user.name "ksamb81"**

**git config --global user.email "ksamb@gmail.com"**

**//**

**git init**

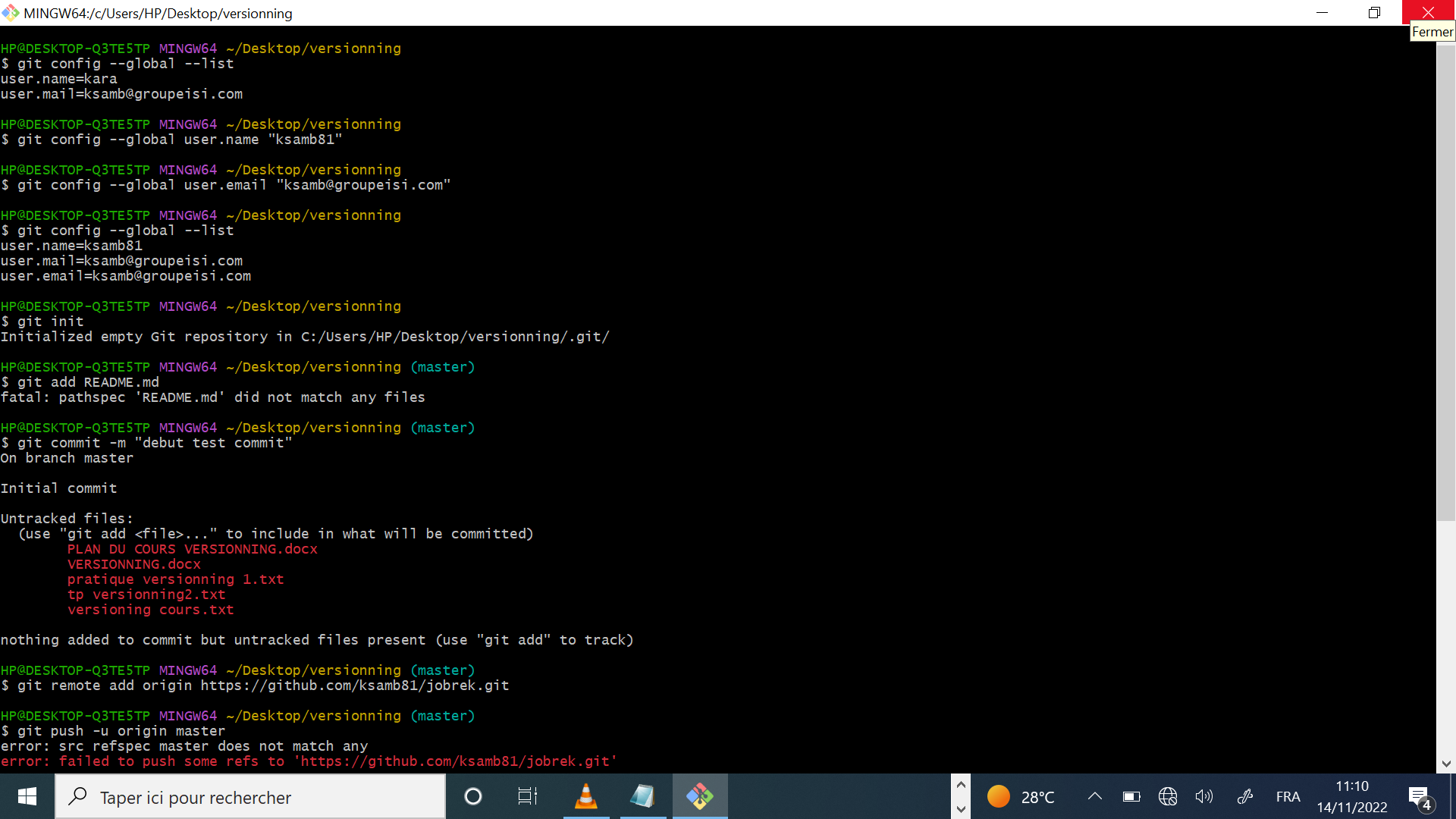
**git add README.md**

**git commit -m “debut commit”**

**git remote add origin** [**https://github.com/ksamb81/jobrek.git**](https://github.com/ksamb81/jobrek.git)

**git push -u origin master**

**Par la pratique**



**NB:** un problème peut surgir sur le push mais néanmoins on peut le régler avec les commandes:

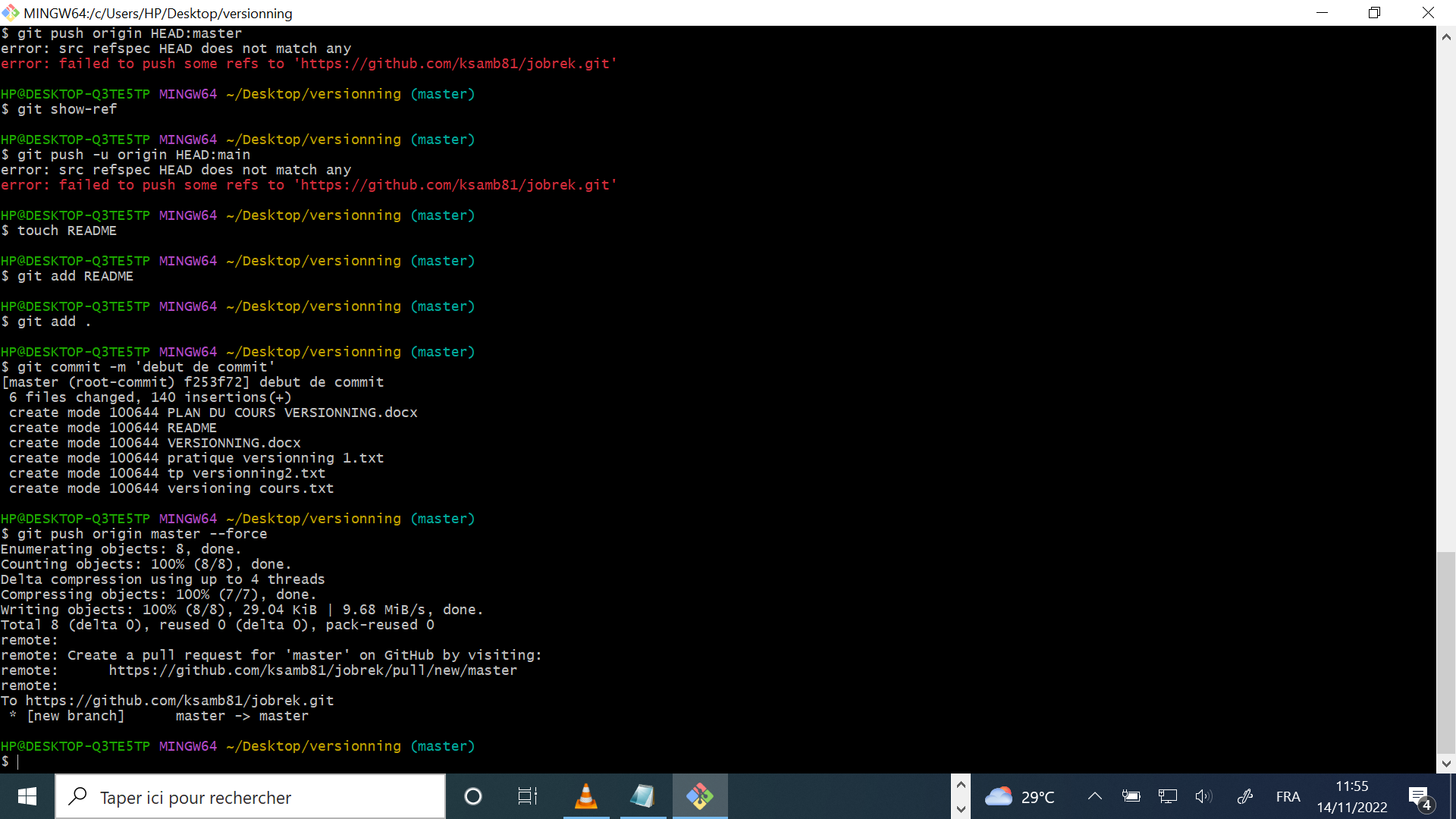
**touch README**

**git add README**

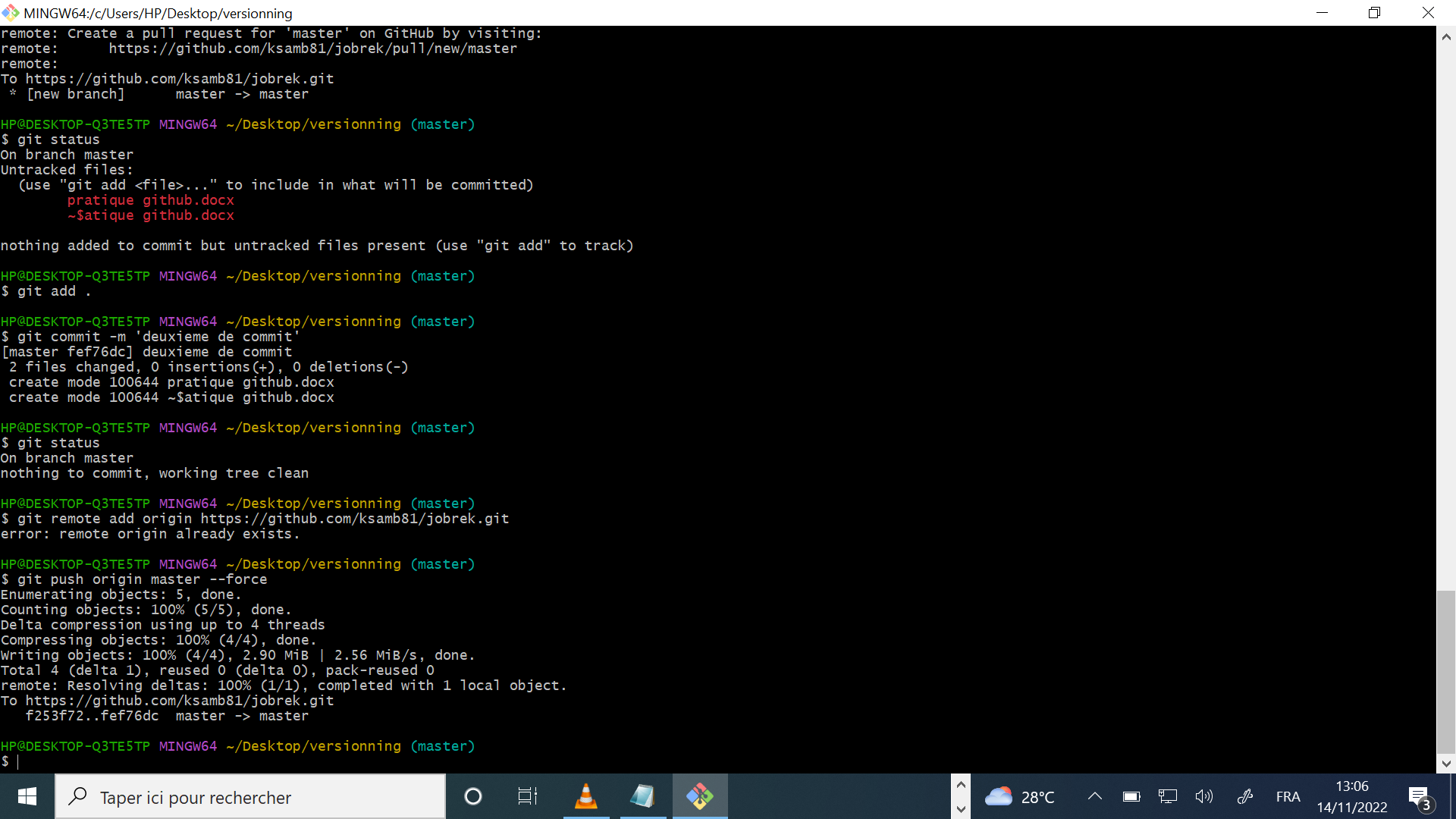
**git add .**

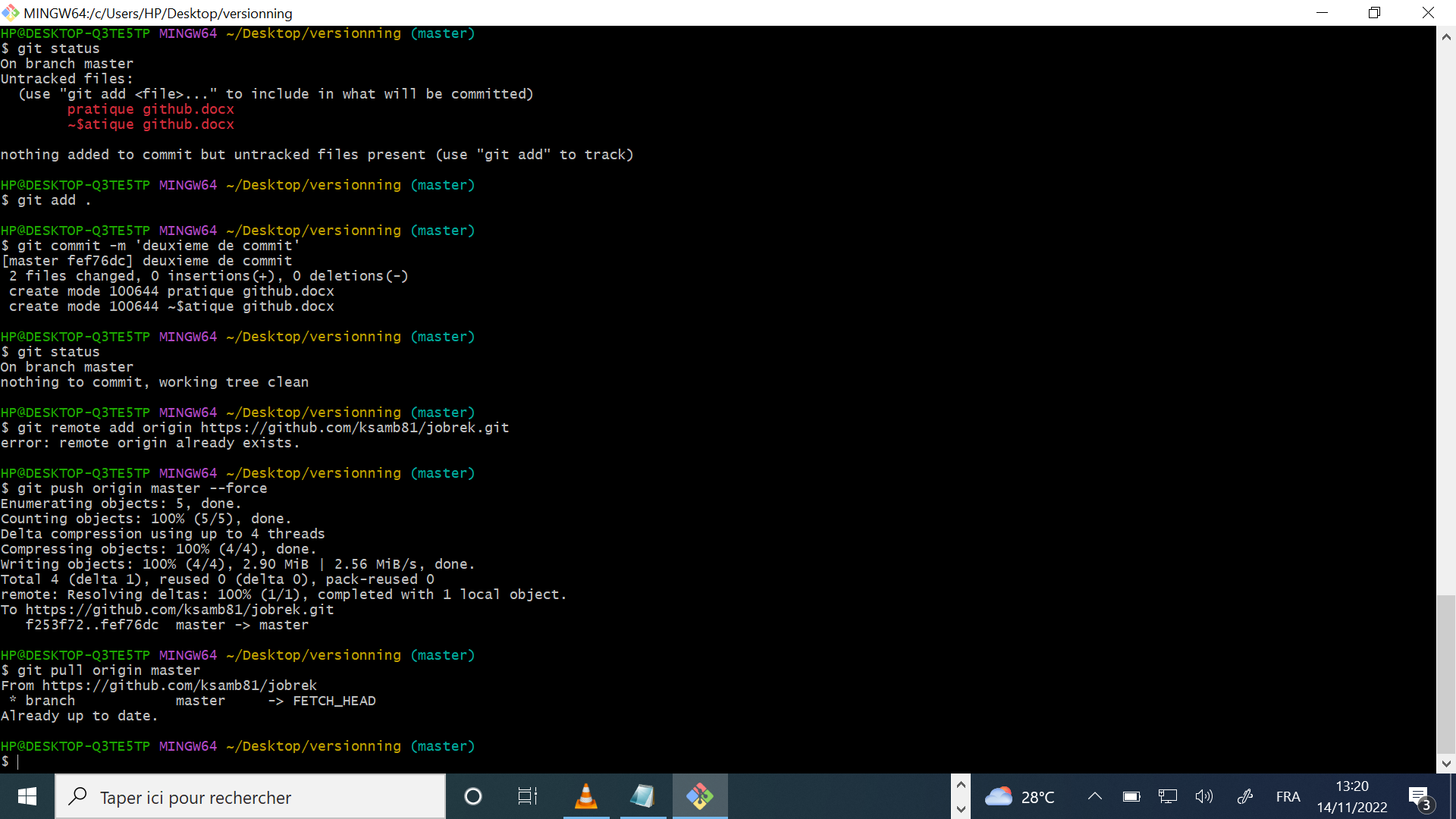
**git commit -m 'reinitialized files'**

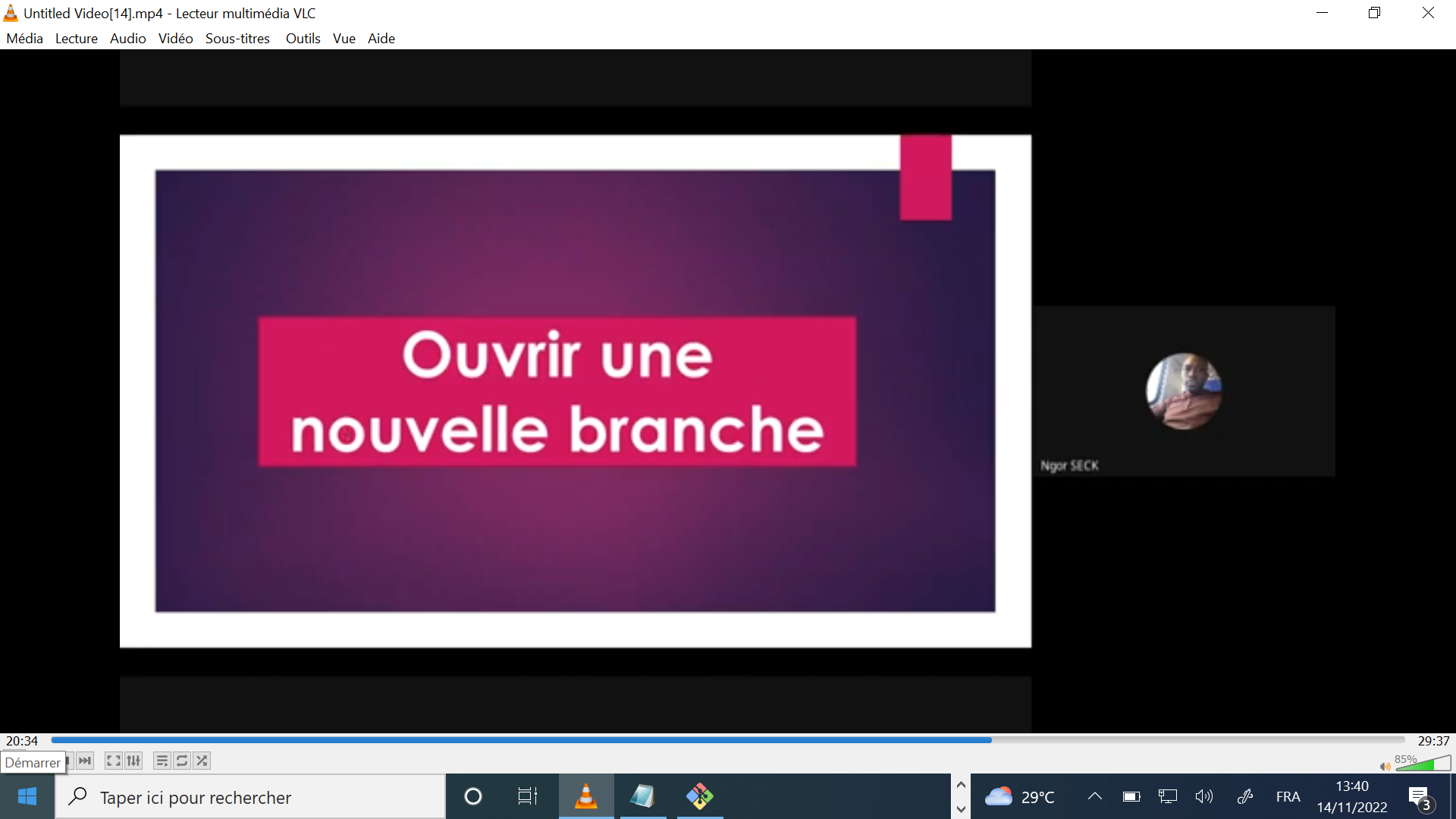
**git push origin master --force**

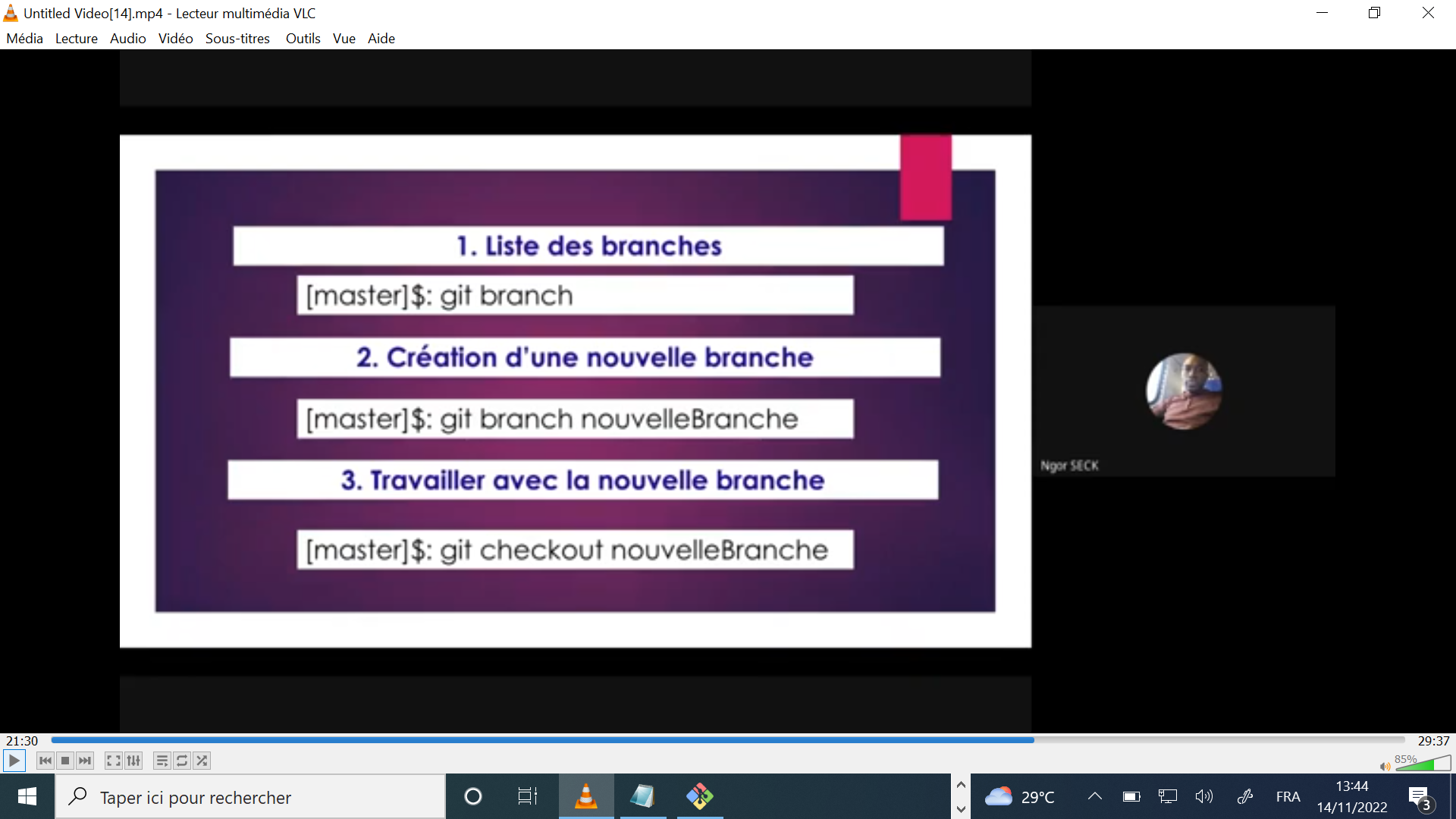


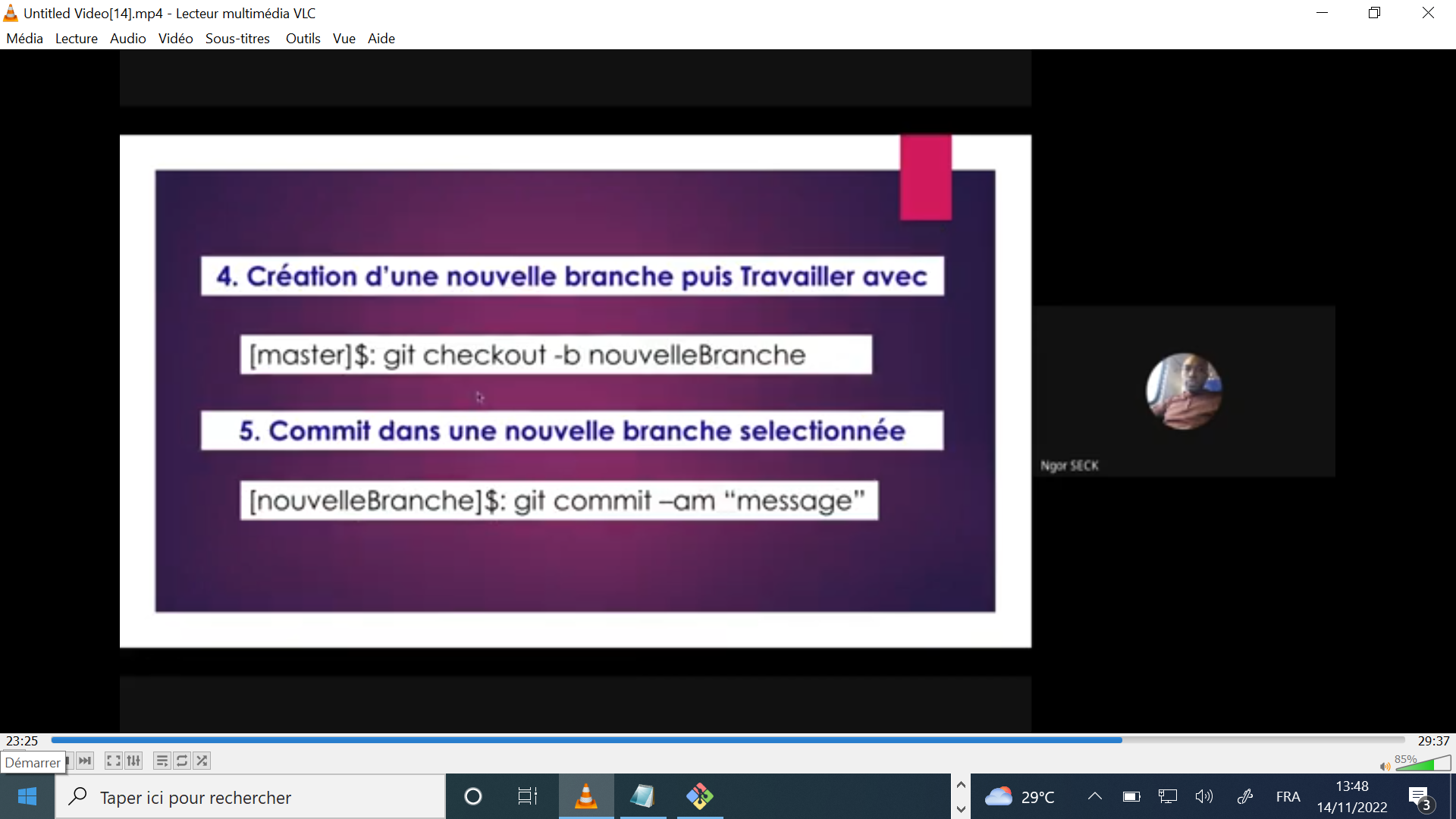
**MODIFICATION**











# Les branches avec Git - Les branches en bref

Presque tous les VCS proposent une certaine forme de gestion de branches. Créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.

Pour de nombreux VCS, il s’agit d’un processus coûteux qui nécessite souvent la création d’une nouvelle copie du répertoire de travail, ce qui peut prendre longtemps dans le cas de gros projets.

Certaines personnes considèrent le modèle de gestion de branches de Git comme ce qu’il a de plus remarquable et il offre sûrement à Git une place à part au sein de la communauté des VCS. En quoi est-il si spécial ? La manière dont Git gère les branches est incroyablement légère et permet de réaliser les opérations sur les branches de manière quasi instantanée et, généralement, de basculer entre les branches aussi rapidement. À la différence de nombreux autres VCS, Git encourage des méthodes qui privilégient la création et la fusion fréquentes de branches, jusqu’à plusieurs fois par jour. Bien comprendre et maîtriser cette fonctionnalité vous permettra de faire de Git un outil puissant et unique et peut totalement changer votre manière de développer.

## Les branches en bref

Pour réellement comprendre la manière dont Git gère les branches, nous devons revenir en arrière et examiner de plus près comment Git stocke ses données.

Si vous vous souvenez bien du chapitre [Démarrage rapide](https://git-scm.com/book/fr/v2/ch00/ch01-getting-started), Git ne stocke pas ses données comme une série de modifications ou de différences successives mais plutôt comme une série d’instantanés (appelés **snapshots**).

Lorsque vous faites un commit, Git stocke un objet **commit** qui contient un pointeur vers l’instantané (**snapshot**) du contenu que vous avez indexé. Cet objet contient également les noms et prénoms de l’auteur, le message que vous avez renseigné ainsi que des pointeurs vers le ou les **commits** qui précèdent directement ce **commit** : aucun parent pour le **commit** initial, un parent pour un **commit** normal et de multiples parents pour un **commit** qui résulte de la fusion d’une ou plusieurs branches.

Pour visualiser ce concept, supposons que vous avez un répertoire contenant trois fichiers que vous indexez puis validez. L’indexation des fichiers calcule une empreinte (**checksum**) pour chacun (via la fonction de hachage SHA-1 mentionnée au chapitre [Démarrage rapide](https://git-scm.com/book/fr/v2/ch00/ch01-getting-started)), stocke cette version du fichier dans le dépôt Git (Git les nomme **blobs**) et ajoute cette empreinte à la zone d’index (**staging area**) :

$ git add README test.rb LICENSE

$ git commit -m 'initial commit of my project'

Lorsque vous créez le **commit** en lançant la commande git commit, Git calcule l’empreinte de chaque sous-répertoire (ici, seulement pour le répertoire racine) et stocke ces objets de type arbre dans le dépôt Git. Git crée alors un objet **commit** qui contient les méta-données et un pointeur vers l’arbre de la racine du projet de manière à pouvoir recréer l’instantané à tout moment.

Votre dépôt Git contient à présent cinq objets : un **blob** pour le contenu de chacun de vos trois fichiers, un arbre (**tree**) qui liste le contenu du répertoire et spécifie quels noms de fichiers sont attachés à quels **blobs** et enfin un objet **commit** portant le pointeur vers l’arbre de la racine ainsi que toutes les méta-données attachées au **commit**.

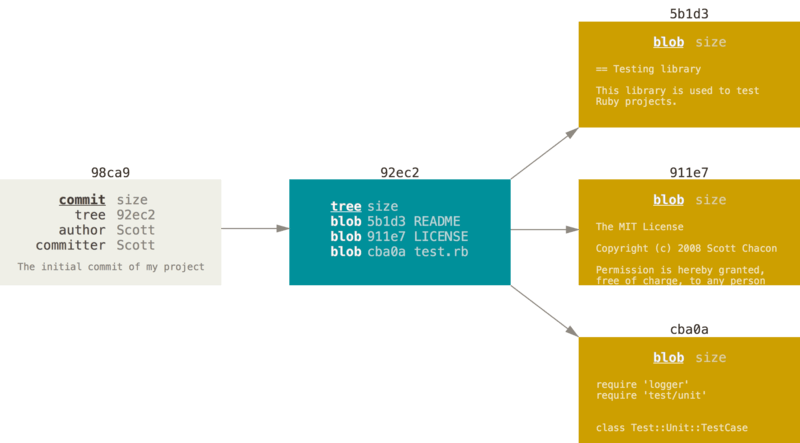


Figure 9. Un commit et son arbre

Si vous faites des modifications et validez à nouveau, le prochain **commit** stocke un pointeur vers le **commit** le précédant immédiatement.

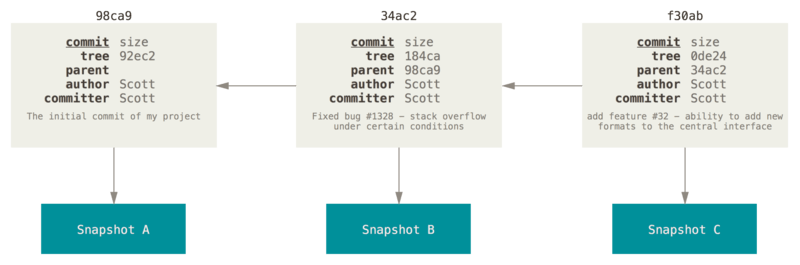


Figure 10. Commits et leurs parents

Une branche dans Git est simplement un pointeur léger et déplaçable vers un de ces **commits**. La branche par défaut dans Git s’appelle master. Au fur et à mesure des validations, la branche master pointe vers le dernier des **commits** réalisés. À chaque validation, le pointeur de la branche master avance automatiquement.

|  |  |
| --- | --- |
| **Note** | La branche master n’est pas une branche spéciale. Elle est identique à toutes les autres branches. La seule raison pour laquelle chaque dépôt en a une est que la commande git init la crée par défaut et que la plupart des gens ne s’embêtent pas à la changer. |

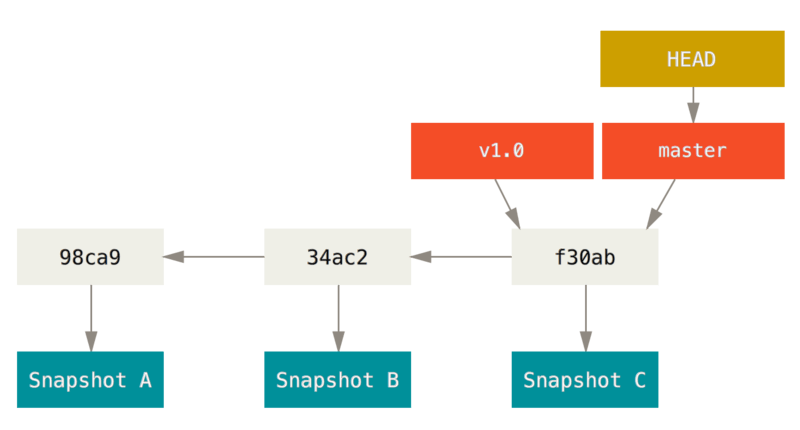


Figure 11. Une branche et l’historique de ses commits

### Créer une nouvelle branche

Que se passe-t-il si vous créez une nouvelle branche ? Eh bien, cela crée un nouveau pointeur pour vous. Supposons que vous créez une nouvelle branche nommée test. Vous utilisez pour cela la commande git branch :

$ git branch testing

Cela crée un nouveau pointeur vers le **commit** courant.

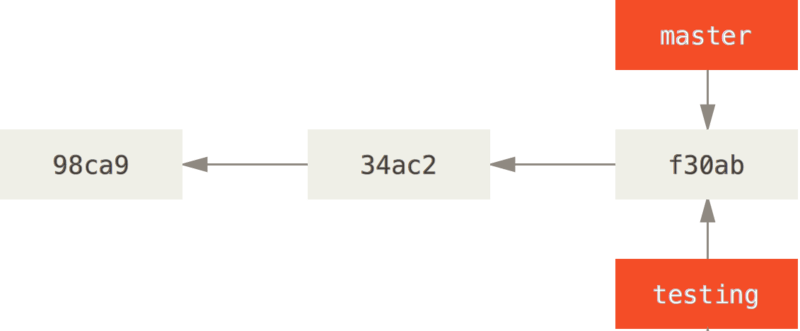


Figure 12. Deux branches pointant vers la même série de commits

Comment Git connaît-il alors la branche sur laquelle vous vous trouvez ? Il conserve à cet effet un pointeur spécial appelé HEAD. Vous remarquez que sous cette appellation se cache un concept très différent de celui utilisé dans les autres VCS tels que Subversion ou CVS. Dans Git, il s’agit simplement d’un pointeur sur la branche locale où vous vous trouvez. Dans ce cas, vous vous trouvez toujours sur master. En effet, la commande git branch n’a fait que créer une nouvelle branche — elle n’a pas fait basculer la copie de travail vers cette branche.

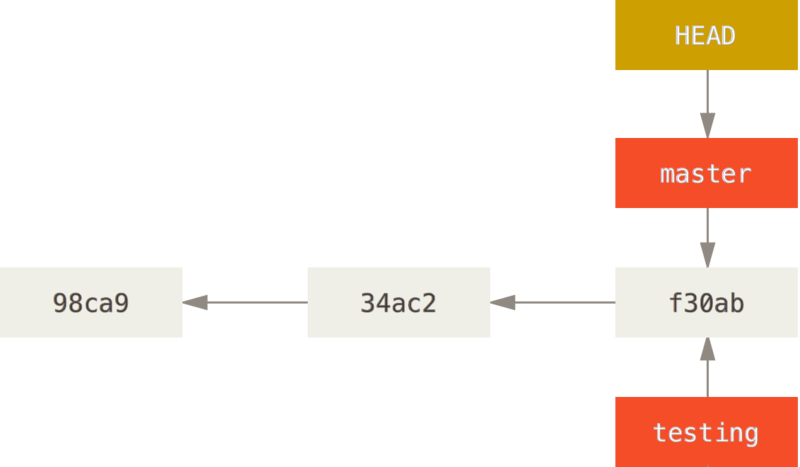


Figure 13. HEAD pointant vers une branche

Vous pouvez vérifier cela facilement grâce à la commande git log qui vous montre vers quoi les branches pointent. Il s’agit de l’option --decorate.

$ git log --oneline --decorate

f30ab (HEAD, master, test) add feature #32 - ability to add new

34ac2 fixed bug #ch1328 - stack overflow under certain conditions

98ca9 initial commit of my project

Vous pouvez voir les branches master et test qui se situent au niveau du **commit** f30ab.

### Basculer entre les branches

Pour basculer sur une branche existante, il suffit de lancer la commande git checkout. Basculons sur la nouvelle branche testing :

$ git checkout testing

Cela déplace HEAD pour le faire pointer vers la branche testing.

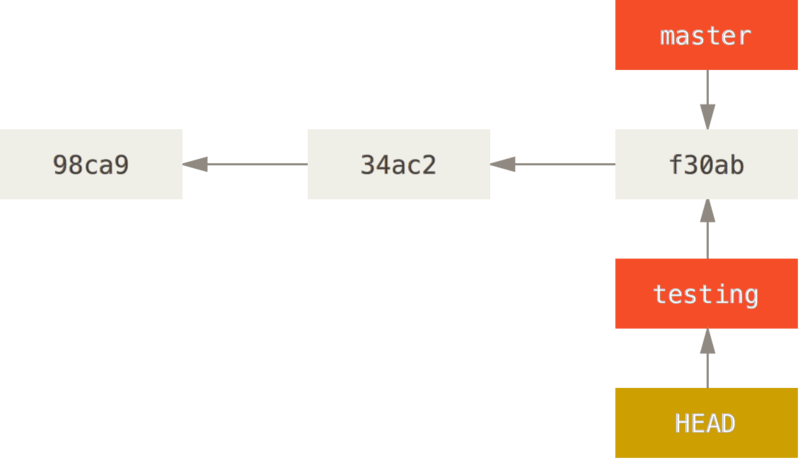


Figure 14. HEAD pointe vers la branche courante

Qu’est-ce que cela signifie ? Et bien, faisons une autre validation :

$ vim test.rb

$ git commit -a -m 'made a change'

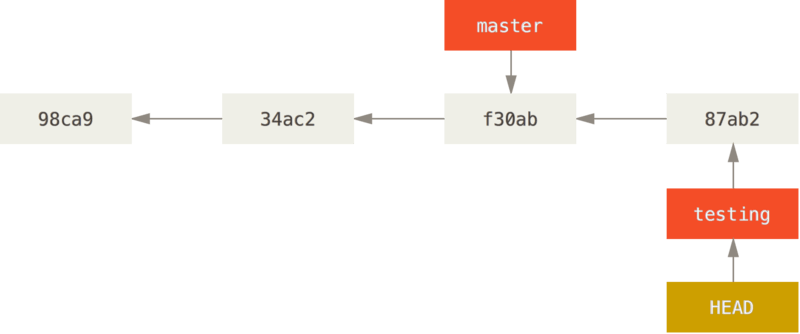


Figure 15. La branche HEAD avance à chaque commit

C’est intéressant parce qu’à présent, votre branche test a avancé tandis que la branche master pointe toujours sur le **commit** sur lequel vous étiez lorsque vous avez lancé la commande git checkout pour changer de branche. Retournons sur la branche master :

$ git checkout master

|  |  |
| --- | --- |
| **Note** | **git log ne montre pas toutes les branches tout le temps**  Si vous alliez lancer git log maintenant, vous pourriez vous demander où se trouve la branche « testing » que vous avez tout juste créée, car elle n’apparaît pas dans la sortie.  La branche n’a pas disparu ; Git ne sait juste pas que cette branche vous intéresse et il essaie de vous montrer ce qu’il pense être le plus pertinent. Autrement dit, par défaut, git log ne montre que l’historique des commits sous la branche qui est extraite.  Pour montrer l’historique des commites de la branche désirée, vous devez la spécifier explicitement : git log testing. Pour afficher toutes les branches, ajoutez l’option --all à la commande git log. |

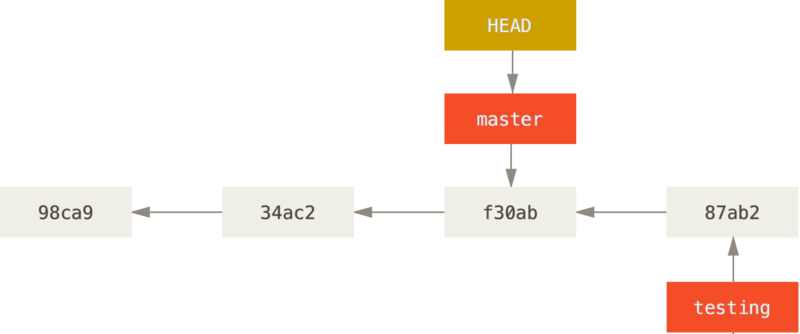


Figure 16. HEAD est déplacé lors d’un checkout

Cette commande a réalisé deux actions. Elle a remis le pointeur HEAD sur la branche master et elle a replacé les fichiers de votre répertoire de travail dans l’état du **snapshot** pointé par master. Cela signifie aussi que les modifications que vous réalisez à partir de ce point divergeront de l’ancienne version du projet. Cette commande annule les modifications réalisées dans la branche test pour vous permettre de repartir dans une autre direction.

|  |  |
| --- | --- |
| **Note** | **Changer de branche modifie les fichiers dans votre répertoire de travail**  Il est important de noter que lorsque vous changez de branche avec Git, les fichiers de votre répertoire de travail sont modifiés. Si vous basculez vers une branche plus ancienne, votre répertoire de travail sera remis dans l’état dans lequel il était lors du dernier **commit** sur cette branche. Si git n’est pas en mesure d’effectuer cette action proprement, il ne vous laissera pas changer de branche. |

Réalisons quelques autres modifications et validons à nouveau :

$ vim test.rb

$ git commit -a -m 'made other changes'

Maintenant, l’historique du projet a divergé (voir [Divergence d’historique](https://git-scm.com/book/fr/v2/ch00/divergent_history)). Vous avez créé une branche et basculé dessus, y avez réalisé des modifications, puis vous avez rebasculé sur la branche principale et réalisé d’autres modifications. Ces deux modifications sont isolées dans des branches séparées : vous pouvez basculer d’une branche à l’autre et les fusionner quand vous êtes prêt. Et vous avez fait tout ceci avec de simples commandes : branch, checkout et commit.

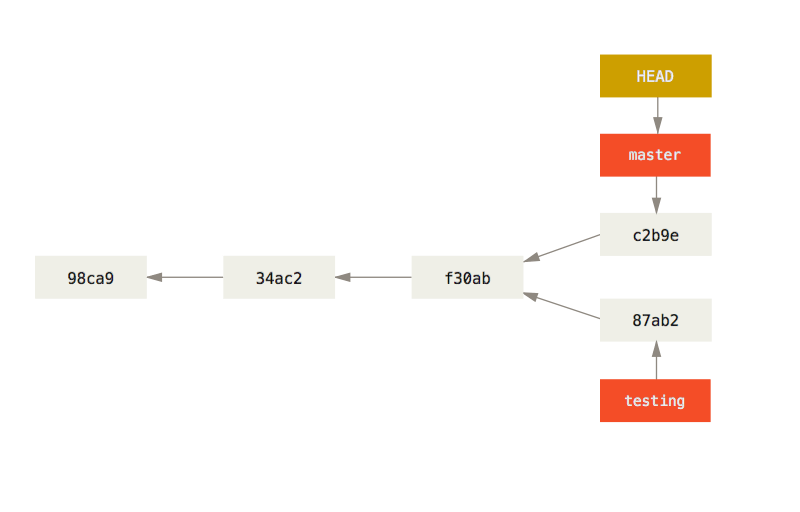


Figure 17. Divergence d’historique

Vous pouvez également voir ceci grâce à la commande git log. La commande git log --oneline --decorate --graph --all va afficher l’historique de vos **commits**, affichant les endroits où sont positionnés vos pointeurs de branche ainsi que la manière dont votre historique a divergé.

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (test) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #ch1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

Parce qu’une branche Git n’est en fait qu’un simple fichier contenant les 40 caractères de l’empreinte SHA-1 du **commit** sur lequel elle pointe, les branches ne coûtent quasiment rien à créer et à détruire. Créer une branche est aussi simple et rapide qu’écrire 41 caractères dans un fichier (40 caractères plus un retour chariot).

C’est une différence de taille avec la manière dont la plupart des VCS gèrent les branches, qui implique de copier tous les fichiers du projet dans un second répertoire. Cela peut durer plusieurs secondes ou même quelques minutes selon la taille du projet, alors que pour Git, le processus est toujours instantané. De plus, comme nous enregistrons les parents quand nous validons les modifications, la détermination de l’ancêtre commun approprié pour la fusion est réalisée automatiquement pour nous et est généralement une opération très facile. Ces fonctionnalités encouragent naturellement les développeurs à créer et utiliser souvent des branches.

Voyons pourquoi vous devriez en faire autant.

|  |  |
| --- | --- |
| **Note** | **Créer une branche et basculer dessus en même temps**  Il est habituel de créer une nouvelle branche et de vouloir basculer sur cette nouvelle branche en même temps — ça peut être réalisé en une seule opération avec git checkout -b <nouvelle-branche>. |
| **Note** | Depuis Git version 2.23, on peut utiliser git switch au lieu de git checkout pour :   * basculer sur une branche existante : git switch testing-branch, * créer une branche et basculer dessus ; git switch -c nouvelle-branche; le drapeau -c signifie créer, vous pouvez aussi utiliser le drapeau complet --create, * revenir sur votre branche précédemment extraite : git switch -. |