Git est un outil puissant, pouvant être simple comme complexe. Ici, nous allons traiter les différentes étapes et situations auxquelles vous pouvez être confrontés lors de son utilisation.

1. Vous avez dit GIT ?

Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. En 2016, il s'agit du logiciel de gestion de versions le plus populaire qui est utilisé par plus de douze millions de personnes.
Pour plus d'informations, consultez cette page (https://openclassrooms.com/fr/courses/1233741-gerez-vos-codes-source-avec-git)

2. Utilisation

Pour cette partie, nous allons nous attarder au fonctionnement général de Git, tel qu'il est actuellement utilisé chez **Subteno IT** pour versionner les projets clients. Dans un premier temps, nous allons voir ce qu'est un dpôt GIT, pour ensuite découvrir les commandes qui permettent de le contrôler et de mieux versionner son code.

2.1 Récupérer un dépôt (git clone)

Pour travailler tous ensembles, et garder un historique des modifications apportées à un projet, l'utilisation de Git s'est avérée très importante.
Lorsque vous développez, vous travailler généralement dans un dossier, qui contient tout le code relatif au projet client, sur lequel vous allez par la suite apporter des modifications.
Ce code est versionné, et pour vous simplifier la vie, vous allez simplement pouvoir vous dire que le dossier, que vous avez en local sur lequel vous travaillez, est en réalité votre **dépôt GIT.**
On le reconnaît notamment grâce au dossier **.git** existant dans le code projet.
Dans notre cas, nous allons voir un cas concret au travers d'un projet Client chez Subteno IT. Ces projets sont disponibles sur Gitlab, qui voit son utilisation documentée ici (gitlab.html#outils-internes-gitlab)

> Warning
>
> Nous ne traiterons pas de Gitlab ici. Pour pouvoir suivre cette documentation, il est nécessaire d'avoir au préalable enregistré sa clé SSH dans Gitlab. La documentation pour le faire **se situe ici (gitlab.html#outils-internes-gitlab-ajout-cle-ssh)** . Dans le cas de cette documentation, nous allons travailler sur le projet client Apple, qui est un exercice.

Pour récupérer un dépôt de projet, il suffit de le **cloner** en local. Cloner un dépôt gitlab revient à récupérer une copie locale d'un dépôt projet (de son code source), en vue d'y apporter des modifications.
Avant de commencer, si ce n'est pas fait, forkez le dépôt **Apple** dans votre espace Gitlab, afin d'en créer une copie. Si vous ne savez pas ce qu'est un fork, vous pouvez retrouver la procédure à cet endroit (gitlab.html#outils-internes-gitlab-fork). Une fois le fork réalisé, vous pouvez ensuite le cloner.
Pour cela, lancez la commande suivante (dans un terminal) dans le dossier **~/Projects/Clients** :

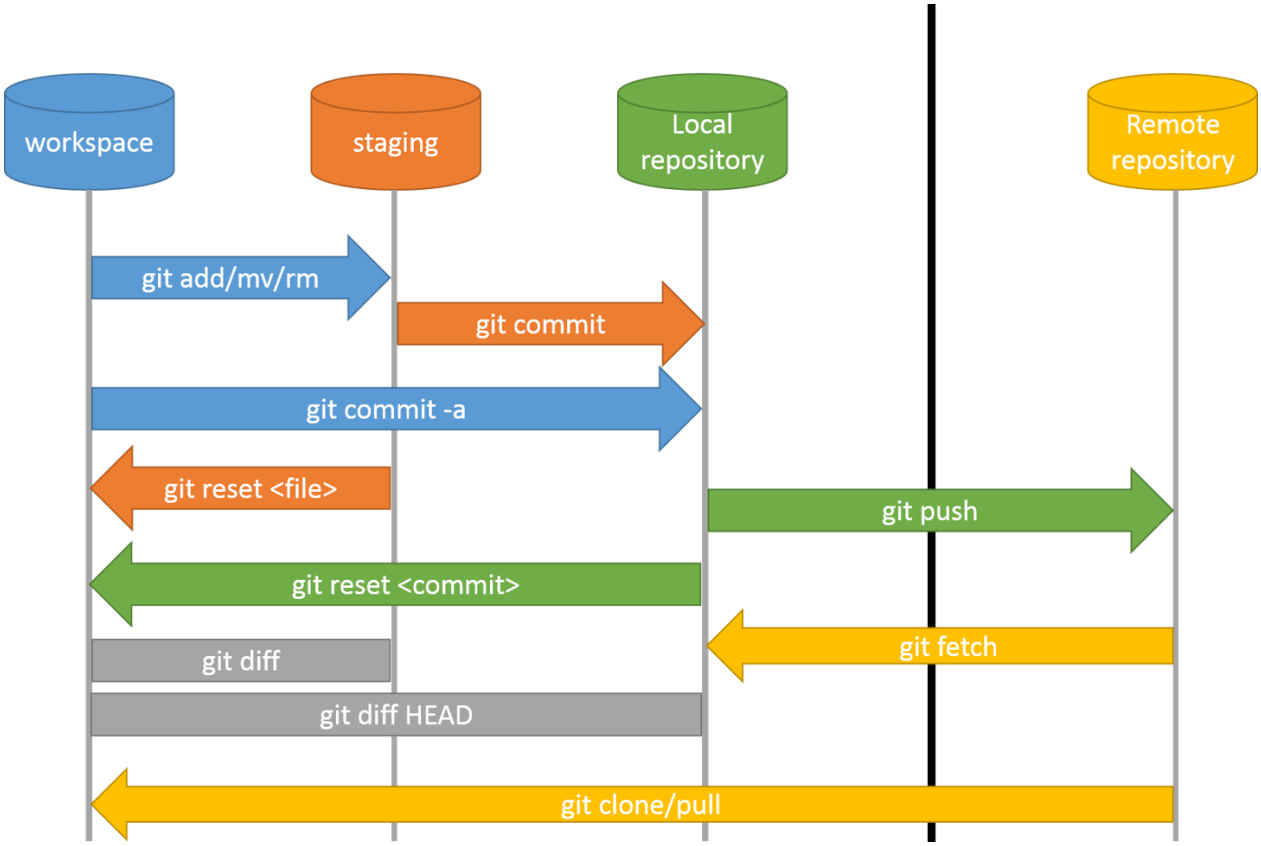
```
git clone <url_ssh_de_votre_fork>
```

> Tip
>
> Le chemin de votre MAC pour accéder au répertoire **~/Projects/Clients** est accessible via la commande **`cdc`** dans un terminal. C'est alias qui est défini pour gagner du temps !
>
> L'URL SSH de votre fork se trouve en cliquant sur le bouton bleu **clone**, dans votre fork du projet.

Maintenant que vous avez cloné votre dépôt de projet, nous allons pouvoir commencer à travailler sur le projet.

2.2 Graphique explicatif de GIT

Dans notre cas, nous allons surtout essayer de comprendre comment fonctionne GIT.
Pour cela, un petit graphique sympathique s'impose :



Ce schéma explique très bien le fonctionnement de Git, lorsque vous travaillez en local !
Pour exempliquer sommairement, lorsque vous travaillez en local, 3 étapes existent au cours de votre développement avec **Git**, schématisée au dessus :

`Workspace` : Correspond à votre dossier seul, sans inclure le répertoire **.git** qui contient vos commits, votre historique des modifications … en bref, votre dépôt git. Le Workspace Correspond donc lui bien à votre espace de travail tel qu'il existe, sans GIT.

`Staging` : Correspond en réalité à l'index Git. L'index Git, c'est l'endroit ou l'on dit à Git : "Fais ci, fais ça". En somme, il est capable dans cet index, de comprendre ce que vous faites, et de vous récapituler les modifications en cours. Pour faire court, il vous dit ici "Qu'est ce qu'il s'est passé depuis le dernier "checkpoint –> commit".

`Local Repository` : C'est votre dépôt git local. Typiquement, il est comme les dépôts clients distants que vous pouvez retrouver sur Gitlab, ou même vos propres forks. Il contient l'historique, et vous permet de travailler. Lorsque vous **commitez** vos modifications, vous les sauvegardez dans votre dépôt local, qui en garde alors une trace.

Enfin, vous pouvez voir une 4ème étape :

`Remote Repository` : Correspond à votre dépôt distant. En somme, **votre fork** distant, sur lequel vous pousserez vos développements par la suite, en vue de faire votre Merge Request dans le dépôt client principal !

2.3 Gestion de l'index GIT

Dans cette documenation, nous allons principalement aborder l'index GIT, le dépôt local ainsi que le dépôt distant. Le Workspace, en soit, ne représente rien d'intéressant, puisqu'il ne fait pas partie intégrante de Git, vu qu'il représente votre dossier, seul, tel qu'il est.
Quand on parle d'index Git, on parle donc de savoir [UNKNOWN NODE problematic]ce que l'on est en train de faire".
Une commande utile existe alors, en terminal, dans le dossier projet :

`git status` (alias Subteno : **`git st`**) : Cette commande vous permet de savoir où vous en êtes actuellement. Un exemple simple, au moment au j'écris ces lignes, voici ce qu'il ressort de cette commande dans mon cas :

```
{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$ git status
On branch 12.0
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:   outils_internes/git.rst

no changes added to commit (use "git add" and/or "git commit -a")
{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$
```

Clairement, ici GIT me dit que j'ai fais des modifications (en effet, j'écris ces lignes !), qui nécessitent d'être commitées ou suivies par la suite. En fait, j'ai simplement sauvegardé un fichier dans lequel j'ai modifié du code. Faites en de même dans votre dépôt local pour vous entrainer !
Pour ajouter mes modifications à l'index Git, et lui dire en somme "Suis ces modifications !", je peux utiliser la commande suivante :
`git add outils_internes/git.rst` : Ajoute le fichier que j'ai modifié à l'index GIT.
Si j'utilise à nouveau la commande **`git status`** , j'obtiens alors ce résultat :

```
{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$ git st
On branch 12.0
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   outils_internes/git.rst

{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$
```

On voit alors un fichier en vert. Git m'indique alors que le fichier est bien dans son index, et qu'il suit les modifications réalisées sur ce fichier.
Si l'on veut annuler cette commande, et ne plus suivre les modifications réalisées, il est possible d'utiliser la commande suivante :
`git reset` : Supprime tout suivi de modification en cours dans l'index GIT.

> Tip
>
> Parfois, vous aurez envie de supprimer intégralement ce que vous avez pu réaliser depuis votre dernier commit (dernière sauvegarde). Pour cela, il est possible d'ajouter un argument à la commande, qui remet tous les fichiers dans leur état au commit antérieur : **`git reset --hard`**

Si vous avez essayé la commande, **ajoutez à nouveau le fichier que vous avez précédemment modifié** pour vous entrainer.
Nous allons maintenant commiter (sauvegarder) les modifications que nous avons faites. Dans un premier temps, nous allons vérifier que nous allons uniquement commiter des changement que nous souhaitons intégrer dans notre prochain commit (je ne parlerai plus de sauvegarde désormais).
Pour cela, utilisez la commande suivante :
`git diff --cached` (alias Subteno **`git di --cached`**) : permet de consulter les modifications que vous avez faites depuis votre dernier commit, qui ne sont pas encore commitées, **mais ajoutées dans votre index Git.**

> Tip
>
> Lorsque vos modifications ne sont pas encore ajoutées à votre index git (via la commande **`git add`**), vos modifications actives sont disponibles via la commande **`git diff`** sans argument.

Dans notre cas, voici le résultat de la première commande :

```
{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$ git di --cached
diff --git a/outils_internes/git.rst b/outils_internes/git.rst
index 0e0e875..95b2db8 100644
--- a/outils_internes/git.rst
+++ b/outils_internes/git.rst
@@ -172,4 +172,4 @@ plus de sauvegarde désormais).

 Pour cela, utilisez la commande suivante :

-
+``git diff --cached`` (alias Subteno ``git di --cached``) :
{12.0} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation-user$
```

Deux informations intéressantes sont alors disponibles :

- Qu'est ce qui est comparé : Une version **A** du fichier, avec sa version **B**. *diff –git a/outils_internes/git.rst b/outils_internes/git.rst*
- Les modifications qui ont été réalisées : Ici, on voit qu'une ligne a été ajoutée dans mon fichier (Grâce au petit ''+'' en début de ligne) par rapport à la précédente version.

Git fonctionne de façon différentielle. Si vous modifiez un mot dans une ligne d'un fichier, et consultez les différences via la commande **`git diff`** , vous retrouverez alors 2 lignes, une vous disant que la ligne modifiée a été retirée, et l'autre stipulant ce qui la remplace.
Nous venons de voir les principales commandes de gestions de l'index Git.

> Tip
>
> Lorsque vous réalisez un développement, vous aurez peut-être besoin, lorsque vous souhaiterez par exemple récupérer les modifications réalisées sur la branche master du projet client, de sauvegarder temporairement votre travail, de le mettre de côté, pour lé récupérer par la suite. La commande **`git stash`** vous le permettra. Une fois cette commande utilisée, vos développements non commités disparaîtront, et pourrons être réappliqués via la commande **`git stash pop`**

2.3 Gestion du dépôt local GIT

Nous avions donc précédemment ajouté nos modifications à l'index GIT. Pour les sauvegarder, nous allons désormais commiter nos modifications pour les sauvegarder définitivement dans l'historique.
Pour committer vos modifications, la commande suivante est disponible :

`git commit -m "[IMP] Improve git documenation"`

L'argument **-m** permet de spécifier par la suite un **message de commit**, qui permet d'identifier simplement ce qui a été fait dans cette version, par rapport à sa version précente.
Chaque commot est un différentiel par rapport au commit qui le précède.
Vous venez ainsi de commiter vos modifications, qui apparaîtront désormais dans l'historiqe GIT.
Pour consulter l'historique, la commande suivante est disponible :

`git log` : Permet d'afficher tout l'historique de commits réalisés sur un projet, afin de le consulter.

> Tip
>
> Une commande plus pratique existe, qui est un alias créé par Subteno IT, pour rendre plus **lisible** l'historique de commits. Essayez la commande **`git llog`** , elle est bien plus intéressante.

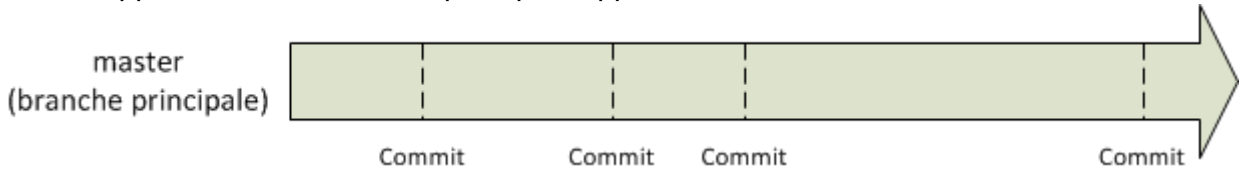
2.4 Envoi de vos modifications sur votre dépôt distant

Une fois vos développements commités, vous avez la possibilité de les envoyer sur votre dépôt distant, situé dans notre cas sur Gitlab.
Pour cela, la commande suivante est disponible :
`git push <remote> <branch>` : Permet de pousser vos modifications sur la branche (<branch>) de votre dépôt distant défini à la place de <remote>. Généralement, le remote correspondant à votre fork se nommera **`òrigin`** .

> Tip
>
> Pensez à ne jamais travailler sur votre branche locale **`master`** . pour cela, vous pouvez créer une branche locale à partir de celle où vous vous situez via la commande suivante : **`git checkout -b nouvelle_branche`**

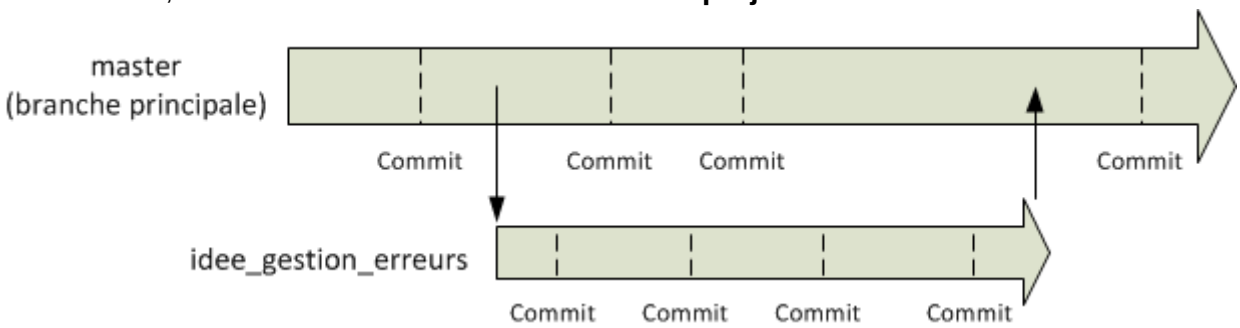
2.5 Principe des branches

Les branches font partie du cœur même de Git et constituent un de ses principaux atouts. C'est un moyen de travailler en parallèle sur d'autres fonctionnalités. C'est comme si vous aviez quelque part une « copie » du code source du projet qui vous permet de tester vos idées les plus folles et de vérifier si elles fonctionnent avant de les intégrer au véritable code source de votre projet.
Bien que les branches soient « la base » de Git, je n'en ai pas parlé avant pour rester simple. Pourtant, il faut absolument les connaître et s'en servir. La gestion poussée des branches de Git est la principale raison qui incite les projets à passer à Git, donc il vaut mieux comprendre comment ça fonctionne et en faire usage, sinon on passe vraiment à côté de quelque chose.
Dans Git, toutes les modifications que vous faites au fil du temps sont par défaut considérées comme appartenant à la branche principale appelée « master » :



On voit sur ce schéma les commits qui sont effectués au fil du temps.
Supposons que vous ayez une idée pour améliorer la logistique dans votre projet mais que vous ne soyez pas sûrs qu'elle va fonctionner ensuite : vous voulez faire des tests, ça va vous prendre du temps, donc vous ne voulez pas que votre projet incorpore ces changements dans l'immédiat.
Il suffit de créer une branche, que vous nommerez par exemple « 3-improve_logistic », dans laquelle vous allez pouvoir travailler en parallèle, depuis votre branche master :
`git checkout -b 3-improve_logistic`

Une branche, c'est en somme **une version de votre projet**.



Une fois que vos modifications seront terminées, il sera alors possible de proposer vos modifications pour qu'elles soient intégrées à la branche **`master`** du projet client. Pour cette partie, elle sera disponible dans la documentation de Gitlab (gitlab.html#outils-internes-gitlab)

> Tip
>
> Vos branches locales sont disponibles par le biais de la commande : **`git branch`**`

3. Commandes utiles

Dans cette section, toutes les commandes intéressantes pouvant être utilisées avec GIT seront listées. Si jamais il vous semble qu'une commande est manquante, parlons-en !

3.1 Ajouter un dépôt externe dans un projet

Exemple générique pour un dépôt hébergé sur GitHub :

```
git subrepo clone -b <branch> git@github.com:
<developer_name>/<repo_name> modules/<developer_name>/<repo_name>
```

> Tip
>
> Si vous n'avez pas git subrepo d'installé sur votre machine vous devez lancer:
>
> ```
> git clone https://github.com/ingydotnet/git-subrepo
> /usr/local/bin/git-subrepo
> echo 'source /usr/local/bin/git-subrepo/.rc' >>
> ~/.bash_profile
> source /usr/local/bin/git-subrepo/.rc
> ```

3.2 Mettre à jour un dépôt externe dans un projet

> Warning
>
> Vous pouvez trouver des anciens projets avec des subtree a lieu de subrepo. Ne vous inquitez pas tout vas bien, dans ce cas pour mettre à jour le dépot externe il faut supprimer le dossier le contenant et suivre la procédure d'ajout (pas la mise à jour).

Exemple générique pour un dépôt hébergé sur GitHub :

```
git subrepo clone -b <branch> git@github.com:
<developer_name>/<repo_name> modules/<developer_name>/<repo_name>
```

Exemple spécifique de la mise à jour du code standard :

```
git subrepo pull server
git subrepo pull modules/odoo/enterprise
```

> Tip
>
> Pour changer un subrepo de branche il faut le supprimer et le re-cloner sur la bonne branche. (Au passage subrepo est un projet opensource donc ouvert à toutes contributions pour régler ce soucis :))

> Tip
>
> Mettre à jour le standard sans mettre à jour les addons enterprise peut poser problème, il est préférable de les garder toujours synchronisés.

3.3 Corriger un commit

Si vous vous rendez compte que vous avez une erreur dans un commit, vous avez 2 possibilités: C'est le dernier commit:

```
git commit --amend
```

C'est un commit plus ancien:

```
git commit --fixup=HASHDUCOMMITACORRIGER git rebase -i --autosquash
```

> Tip
>
> –autosquash n'est pas obligatoire s'il est déjà dans la configuration: `git config rebase.autosquash true`

3.4 Rebaser une branche

Rebaser une branche a pour but de ré-écrire l'historique GIT du dépôt. De cette façon, il est possible de manipuler l'historique à sa guise, en vue de l'arranger.
Par exemple, pour ré-écrire l'historique de vos 5 derniers commits, la commande suivante peut être utilisée pour faire **un rebase interactif** :

```
git rebase -i HEAD~5
```

> Warning
>
> La ré-écriture d'un historique entraine une erreur lors d'un `git push` , si le précédent historique était détenu par le dépôt distant. Il refusera alors toute modification, par précaution, vous notifiant que vous risqueriez de perdre l'historique. Pour forcer le push, vous pouvez ajouter l'argument `-f` à votre commande : `git push -f origin ma_branche`

Concernant le rebase, lorsque vous lancez la commande, un éditeur en ligne de commande (VIM, pour les anciens) va s'ouvrir. Il vous permet de modifier un fichier qui sera lu par GIT pour effectuer son rebase.
Son fonctionnement y est expliqué.

3.5 Utiliser le reflog (Rebase MERDIQUE)

Nooooon ! Ah sacrée boulette, vous venez de ré-écrire votre historique local git, que vous n'aviez auparavant pas poussé, et avez supprimé un commit par mégarde qui contenait 1 bonne heure de travail.
Heureusement, ce n'est pas la fin !
Git conserve un historique complet de ce que vous faites (en plus de l'historique de commits), qui s'appelle le **reflog**.
Comment l'utiliser ? c'est simple, la commandes est la suivante :

```
git reflog
```

Vous pouvez alors séléctionner le Hash correspondant au moment dans l'historique où vous souhaitez revenir, et faire un checkout de ce hash : `git checkout <hash>`

> Tip
>
> Un checkout permet de changer, de branche, de revenir à un commit précis, ou meme de se balader dans le reflog. Pour le coup, c'est une des commandes GIT les plus utiles, surtout en cas de mauvaise manip' !

3.5 Git bisect ! (Allez ! On teste !)

Vous ne la connaissiez pas ? Cette commande est top. Git bisect est une commande Git, qui vous permet d'identifier rapidement le commit responsable d'une régression constatée quelques temps après plusieurs autres commits.
Dans les faits, cette commande fait une recherche **dichotomique** du commit contenant l'erreur, et ce à votre place. Enfin, en partie !
Pour démarrer :

```
git bisect start
```

Ensuite, il suffit de dire à GIT quel commit est bon (tout fonctionne), et quel commit est mauvais. Par la suite, il changera automatiquement de version jusqu'à vous permettre de retrouver **LE** commit qui introduit la régression.
Pour déclarer le dernier commit pour lequel vous savez que le bug n'est pas présent :

```
git bisect good <hash_du_commit>
```

Pour celui qui est mauvais, ou le bug est constaté :

```
git bisect bad <hash_du_commit>
```

Ensuite, laissez-vous porter par la magie de GIT, et testez simplement votre Odoo.
Si le bug est présent, dites le à GIT :

```
git bisect bad`
```

Sinon :

```
git bisect good`
```

Jusqu'à ce qu'il vous dise qu'il a trouvé le commit que vous cherchiez :) Il vous dira, à chaque fois, combien de recherches sont encore nécessaires.
Pour quitter le mode **git bisect**, utilisez la commande suivante :

```
git bisect reset
```

> Tip
>
> Pour utiliser cette commande, assurez vous de ne pas avoir de modifications en cours, et que votre index git est vide via un `git status` .