

Les bases de Git

Table des matières

I. Contexte	3
II. Initialiser un dépôt Git	3
III. Exercice : Appliquez la notion	6
IV. Les modifications en cours	6
V. Exercice : Appliquez la notion	9
VI. Manipuler des fichiers	10
VII. Exercice : Appliquez la notion	13
VIII. Ignorer des fichiers	14
IX. Exercice : Appliquez la notion	16
X. Comparer des changements	16
XI. Exercice : Appliquez la notion	18
XII. Valider une modification	19
XIII. Exercice : Appliquez la notion	21
XIV. Auto-évaluation	21
A. Exercice final	21
B. Exercice : Défi	23
Solutions des exercices	24

I. Contexte

Durée : 1 h

Environnement de travail : GitBash / Terminal

Pré-requis : Avoir suivi le cours d'introduction à Git

Contexte

Nous le savons désormais, en tant que développeurs, lorsque nous travaillerons sur un projet, il conviendra de conserver le code de celui-ci au sein d'un espace partagé. Cela permettra de mettre en place et de sécuriser l'historique du travail effectué, tout en le rendant accessible aux autres membres de l'équipe afin de permettre un travail collaboratif.

À travers ce cours, nous allons voir le cheminement global permettant la mise en œuvre de cette solution. La première étape, indispensable, correspond à la mise en place d'un dépôt. Avec l'avancement du projet se posera la question de savoir comment visualiser les modifications, les ajouter, mais aussi pouvoir les ignorer. En effet, certains fichiers (tels que des fichiers de configuration) n'ont pas à être diffusés. Une fois les modifications identifiées viendra le temps de les valider définitivement.

II. Initialiser un dépôt Git

Objectifs

- Initialiser un dépôt Git
- Configurer un dépôt Git

Mise en situation

Lorsque l'on démarre sur un projet, qu'il s'agisse d'un nouveau projet ou d'un projet déjà existant, la première étape fondamentale consiste en l'initialisation de son dépôt Git sur son environnement local.

Selon le cas, l'initialisation du dépôt se fera alors de deux manières.

Méthode

Initialiser un nouveau dépôt

Pour initialiser un nouveau dépôt, par exemple dans le cadre d'un nouveau projet, il suffit de se placer dans le répertoire qui contiendra les sources de ce projet et d'exécuter la commande `git init`.

Lorsque cette commande est exécutée, un nouveau répertoire **.git** est créé. Celui-ci contient l'ensemble des éléments nécessaires au bon fonctionnement d'un dépôt Git.

Exemple

On démarre un nouveau projet de refonte d'un intranet. Voici les étapes à effectuer :

1. On crée le répertoire qui contiendra les sources du projet
2. On se déplace dans ce répertoire
3. On initialise le dépôt

```

1 user@user MINGW64 /e/workspace
2 $ mkdir refonte-intranet
3
4 user@user MINGW64 /e/workspace
5 $ cd refonte-intranet/
6
7 user@user MINGW64 /e/workspace/refonte-intranet
8 $ git init
9 Initialized empty Git repository in E:/workspace/refonte-intranet/.git/
10
11 user@user MINGW64 /e/workspace/refonte-intranet (main)
12

```

Conseil Quand initialiser son dépôt ?

L'initialisation du dépôt devrait être la première étape de tout nouveau projet. Cela permet de disposer d'un travail qui est en permanence versionné.

Néanmoins, il ne s'agit que d'une recommandation : il reste tout à fait possible d'initialiser un dépôt à n'importe quelle étape du projet, si le nécessaire n'a pas été fait au préalable. Que le répertoire ciblé soit vide ou non, le processus d'initialisation sera le même.

Complément La branche par défaut

Git est basé sur un système de branches. Par défaut, lorsqu'un nouveau dépôt est initialisé, une nouvelle branche est automatiquement créée, intitulée `main`.

Méthode Cloner un dépôt existant

S'il est fréquent de commencer un projet de zéro, il l'est tout autant de rejoindre une équipe pour travailler sur un projet existant. Celui-ci aura déjà vécu, d'autres personnes auront déjà travaillé sur celui-ci et des sources seront disponibles.

Ainsi, pour récupérer le projet sur son environnement local, il conviendra de disposer de l'URL de son dépôt distant, c'est-à-dire le dépôt qui centralise l'ensemble du travail de l'équipe.

Une fois cette URL récupérée, il convient de se placer dans son répertoire de travail et d'exécuter la commande `git clone [url]`.

Un répertoire sera alors créé automatiquement. Celui-ci sera par défaut nommé de la même façon que dans le répertoire distant.

Si l'on souhaite configurer un autre nom, il convient de l'indiquer au moment de la commande `git clone [url] [nouveau_nom]`.

Syntaxe

```

1 $ git clone url.git
2

```

Exemple

Nouveau contributeur sur un projet, on souhaite récupérer les sources :

```
1
2 $ ls -la
3 total 4
4 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ./
5 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ../
6
7 $ git clone https://github.com/thedaviddias/Front-End-Checklist.git
8 Cloning into 'Front-End-Checklist'...
9 ...
10
11
12 $ ls -la
13 total 8
14 drwxr-xr-x 1 user 1049089 0 mai 7 10:25 ./
15 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ../
16 drwxr-xr-x 1 user 1049089 0 mai 7 10:25 Front-End-Checklist/
17
```

Si l'on souhaite nommer le projet différemment :

```
1 $ ls -la
2 total 4
3 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ./
4 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ../
5
6 $ git clone https://github.com/thedaviddias/Front-End-Checklist.git checklist
7 Cloning into 'checklist'...
8 ...
9
10 $ ls -la
11 total 8
12 drwxr-xr-x 1 user 1049089 0 mai 7 10:25 ./
13 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ../
14 drwxr-xr-x 1 user 1049089 0 mai 7 10:25 checklist/
15
```

Syntaxe **À retenir**

- Pour un **nouveau projet**, on peut initialiser celui-ci grâce à la commande `git init`.
- Pour un **projet existant**, il convient de disposer de l'URL de celui-ci et de la télécharger grâce à la commande `git clone`.

Complément

Démarrer un dépôt Git¹

Les entrailles de Git²

1 <https://git-scm.com/book/fr/v2/Les-bases-de-Git-D%C3%A9marrer-un-d%C3%A9p%C3%B4t-Git>

2 <https://git-scm.com/book/fr/v2/Les-tripes-de-Git-Plomberie-et-porcelaine#ch10-git-internals>

III. Exercice : Appliquez la notion

Vous démarrez un nouveau projet intitulé "Cuisiner Facile" : il s'agira d'une application proposant des recettes de cuisine collaboratives.

Question 1

[solution n°1 p.25]

Créez un répertoire intitulé "cuisiner-facile" et initialisez un dépôt Git dans ce répertoire.

Nouveau contributeur sur un projet mettant à disposition une check-list des bonnes pratiques de gestion des performances, vous avez besoin de récupérer ce projet sur votre environnement local.

Question 2

[solution n°2 p.25]

Initialisez un nouveau dépôt en clonant le projet suivant : <https://github.com/thedaviddias/Front-End-Performance-Checklist.git>.

IV. Les modifications en cours

Objectifs

- Comprendre le cycle de vie des fichiers
- Identifier l'état des fichiers
- Comprendre la notion d'espaces
- Préparer des fichiers à être sauvegardés

Mise en situation

Une fois le dépôt initialisé, le travail sur un projet peut commencer. Nous le savons, il est en partie important de disposer de fichiers versionnés afin de pouvoir travailler en toute sécurité. Ainsi, lorsqu'un projet avance et s'il est dans un état stable, il est primordial de sauvegarder cet état.

Cependant, avant d'effectuer cette sauvegarde, il est important d'apprendre comment vérifier l'état des modifications apportées.

Fondamental Le cycle de vie des fichiers

Il est fondamental de bien visualiser le cycle de vie des fichiers dans Git.

Tout d'abord, ceux-ci peuvent se trouver dans deux états :

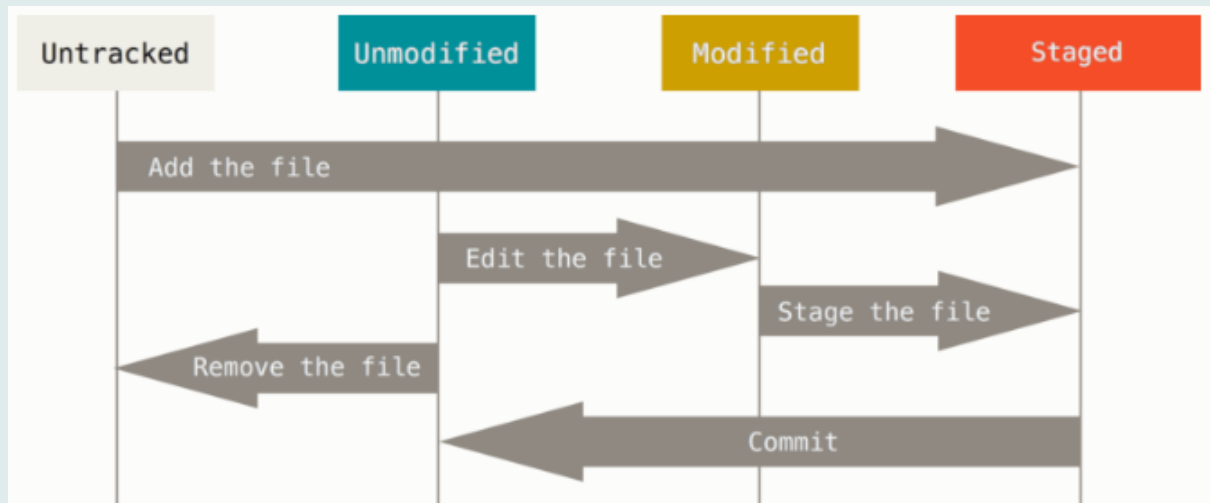
- `Sous suivi de version`, ou `tracked` : il s'agit des fichiers qui étaient déjà présents lors du dernier état des lieux du système,
- `Non-suivi`, ou `untracked` : à l'inverse, il s'agit des fichiers qui n'étaient pas présents lors du dernier état des lieux ou qui ont été supprimés depuis celui-ci.

Par défaut, lors de l'initialisation d'un nouveau dépôt, l'ensemble des fichiers présents seront considérés comme non-suivis, jusqu'à ce qu'une action indique le contraire.

Lorsqu'un fichier est suivi, celui-ci peut se trouver dans trois états différents, à savoir :

- `Non modifié`, ou `unmodified` : le fichier est suivi par Git, mais n'a pas été modifié depuis le dernier état des lieux,
- `Modifié`, ou `modified` : le fichier est suivi par Git et a été modifié depuis le dernier état des lieux,
- `Indexé`, ou `staged` : le fichier a été modifié et ces modifications ont été indexées, mais pas encore enregistrées depuis le dernier état des lieux. Si l'on effectue un `commit`, c'est-à-dire une sauvegarde, seuls les fichiers indexés seront sauvegardés.

Voici comment ces espaces peuvent être représentés schématiquement :



Méthode Consulter l'état de l'espace de travail

Afin de vérifier l'état des modifications apportées, il existe une commande appelée `git status`.

Cette commande permet de visualiser les fichiers non-suivis, modifiés ou indexés. Elle donne également des informations à l'utilisateur sur la façon dont il doit s'y prendre pour modifier l'état des fichiers.

Exemple

On dispose d'un projet nouvellement créé. Un premier commit a permis l'initialisation du dépôt et d'un fichier **.gitignore**.

Depuis ce commit, le projet a continué d'avancer. Si l'on exécute la commande `git status`, on constate :

- Qu'un nouveau répertoire **doc/** a été créé, mais n'a pas encore été suivi, il est **untracked**,
- Que le fichier **.gitignore** a été **modifié**, mais que les modifications apportées ne sont pas encore indexées,
- Que le fichier **README.txt** est un nouveau fichier que l'on a déjà **indexé**, prêt à être sauvegardé.

Pour chaque situation, on constate également que des indications permettent de modifier l'état des fichiers.

```

1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     new file:   README.txt
8
9 Changes not staged for commit:
10    (use "git add <file>..." to update what will be committed)
11    (use "git restore <file>..." to discard changes in working directory)
12     modified:   .gitignore
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16     doc/
17
18
```

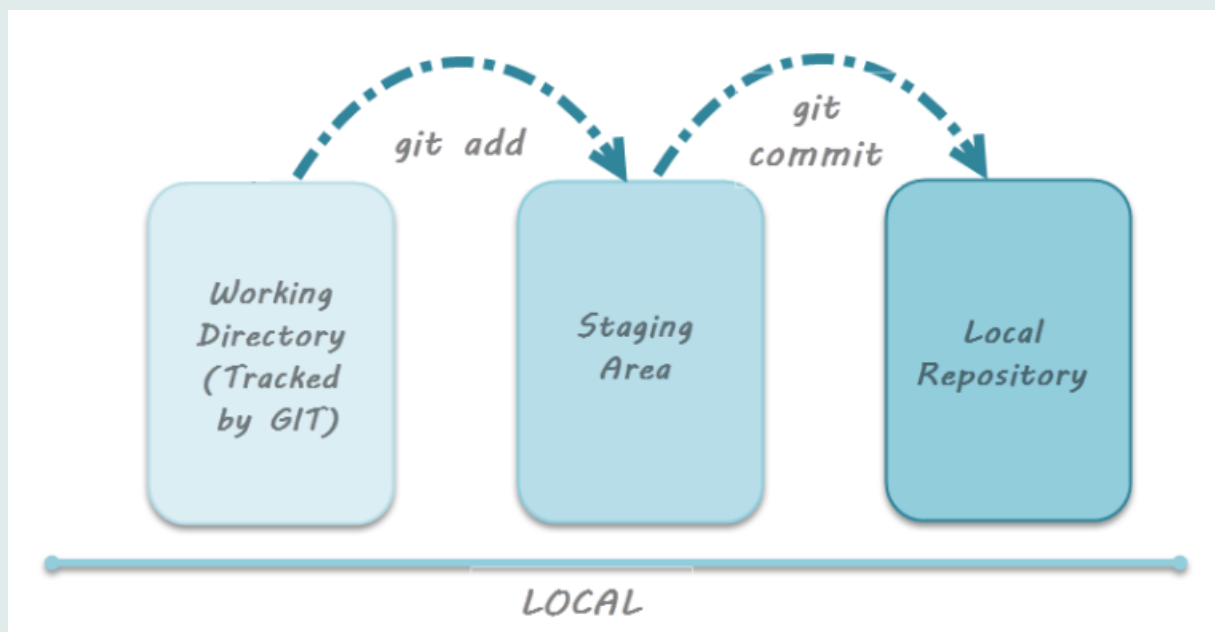
Fondamental Les arborescences de Git

Dans un dépôt Git, on considère qu'il existe différentes arborescences (ou zones) par lesquelles transitent les fichiers au cours de leur cycle de vie.

Ces zones sont des structures de données basées sur des pointeurs et des nœuds, sur lesquels Git se base afin de suivre la chronologie d'édition des fichiers.

- **Le répertoire de travail** : cette zone est synchronisée avec le système de fichiers local. Elle représente l'ensemble des changements apportés au contenu des fichiers et répertoires depuis le dernier commit. Cette zone contient à la fois les fichiers modifiés et les non-suivis, mais non-indexés.
- **La zone de staging, ou zone de transit** : avant que l'on puisse valider définitivement des modifications, il est nécessaires de les faire transiter par cette zone. Cela permet d'être sélectif et de n'enregistrer ce qui est nécessaire qu'au moment désiré. Cette zone contiendra donc l'ensemble des fichiers modifiés depuis le dernier commit. Cela correspond donc aux fichiers ayant été modifiés ou nouvellement créés, qui ont été indexés.
- **Le dépôt local** : cette zone représente l'ensemble des données du dépôt. Elle est matérialisée par le répertoire `.git` et son contenu. Dès lors qu'une action est effectuée (création d'un commit, nouvelle branche), une nouvelle entrée y est ajoutée. Si le répertoire de travail représente le projet à un instant T, le dépôt local représente quant à lui l'ensemble des actions qui y ont été effectuées.

Voici comment l'on pourrait schématiser ces zones :



Méthode Ajouter des fichiers modifiés

Pour ajouter un changement dans le répertoire de travail à la zone de staging, il est nécessaire d'utiliser la commande `git add`.

Cette commande informe Git que l'on souhaite inclure les mises à jour effectuées dans la prochaine sauvegarde de l'espace de travail, c'est-à-dire le prochain commit. Cependant, le fait d'utiliser la commande `git add` n'impacte pas le dépôt de manière significative. Les modifications ne seront enregistrées que lorsque la commande `git commit` aura été exécutée.

Exemple

Reprenons notre exemple précédent. Nous l'avons laissé dans cet état :

```
1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     new file:   README.txt
8
9 Changes not staged for commit:
10    (use "git add <file>..." to update what will be committed)
11    (use "git restore <file>..." to discard changes in working directory)
12      modified:   .gitignore
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16     doc/
17
18
```

On considère que les modifications apportées au fichier **.gitignore** sont correctes, et on décide qu'elles pourront être ajoutées au prochain commit que l'on va effectuer. On joue alors la commande `git add .gitignore`.

Voici ce que l'on obtient si l'on relance la commande `git status` :

```
1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3 On branch main
4 Changes to be committed:
5   (use "git restore --staged <file>..." to unstage)
6     modified:   .gitignore
7     new file:   README.txt
8
9 Untracked files:
10  (use "git add <file>..." to include in what will be committed)
11    doc/
12
```

On constate que le fichier **.gitignore** sera présent dans le prochain commit que l'on effectuera.

Syntaxe **À retenir**

- On vérifie l'état des modifications en cours grâce à la commande `git status`.
- Tout nouveau fichier créé depuis le dernier commit est considéré comme **non-suivi**.
- Tout fichier modifié depuis le dernier commit est automatiquement considéré comme **suivi**.
- Pour que des modifications soient enregistrées dans un commit, celles-ci doivent au préalable être ajoutées à la zone de staging. C'est le rôle de la commande `git add`.

V. Exercice : Appliquez la notion

Lors de l'exercice précédent, vous avez créé un dépôt intitulé **cuisiner-facile**.

Vous conserverez les modifications apportées lors de cet exercice.

Question 1

[solution n°3 p.25]

Créez un fichier **README** dans lequel vous indiquerez "Initialisation du fichier". Vérifiez que ce fichier est bien présent dans la liste des fichiers non-suivis.

Question 2

[solution n°4 p.25]

Ajoutez ce fichier à la liste des fichiers suivis/indexés et vérifiez que celui-ci en fait désormais partie.

VI. Manipuler des fichiers

Objectifs

- Annuler des modifications effectuées sur un fichier
- Retirer un fichier de la zone de transit
- Déplacer ou supprimer un fichier

Mise en situation

Le besoin de manipuler certains fichiers peut rapidement se faire sentir. Comment revenir en arrière si un fichier a été ajouté à la zone de staging par mégarde ? Ou si l'on souhaite tout simplement annuler les modifications effectuées sur un élément ?

Dans certains cas, il peut aussi être utile de déplacer ou supprimer un fichier. Git propose des commandes afin de réaliser cela de façon optimisée.

Méthode Annuler les modifications effectuées sur un fichier

Il peut s'avérer utile de savoir comment annuler les modifications effectuées sur un fichier, afin de rétablir son état par rapport au dernier commit en date. Pour cela, il existe une commande appelée `git restore`¹.

On l'utilise en indiquant le fichier ou répertoire ciblé, à savoir : `git restore element_cible`.

Exemple

Admettons qu'après avoir effectué la commande `git status`, nous nous rendions compte que le fichier **.gitignore** n'aurait pas dû être modifié.

```
1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     new file:   README.txt
8
9 Changes not staged for commit:
10    (use "git add <file>..." to update what will be committed)
11    (use "git restore <file>..." to discard changes in working directory)
12     modified:   .gitignore
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16     doc/
17
18 # Le contenu actuel du fichier
```

¹ <https://git-scm.com/docs/git-restore>

```
19 $ cat .gitignore
20 *.pdf
```

Il suffit alors d'exécuter la commande `git restore .gitignore` pour annuler les modifications effectuées sur ce fichier.

Si l'on vérifie de nouveau l'état de nos fichiers, on constate que celui-ci n'apparaît plus :

```
1 # Le fichier n'est plus suivi
2 $ git status
3 On branch main
4 Changes to be committed:
5   (use "git restore --staged <file>..." to unstage)
6     new file:   README.txt
7
8 Untracked files:
9   (use "git add <file>..." to include in what will be committed)
10    doc/
11
12 # Le fichier est vide, tel qu'il l'était initialement
13 $ cat .gitignore
14
```

Méthode Retirer un fichier de la zone de transit

Il est possible qu'un fichier ait été ajouté à la zone de transit par erreur. Pour cela, il est possible d'utiliser la commande `git restore` en lui associant le paramètre `--staged`.

Si cette commande est déclenchée avec ce paramètre, alors le fichier reviendra dans les fichiers modifiés/non-suivis, selon son état initial. En revanche, les modifications apportées à ce fichier ne seront pas perdues.

Exemple

Admettons que nos fichiers soient dans l'état suivant :

```
1 $ git status
2 On branch main
3 Changes to be committed:
4   (use "git restore --staged <file>..." to unstage)
5     modified:   .gitignore
6     new file:   README.txt
7
8 Untracked files:
9   (use "git add <file>..." to include in what will be committed)
10    doc/
11
12 # Le contenu actuel du fichier
13 $ cat .gitignore
14 *.pdf
15
```

On ne souhaite pas que **.gitignore** soit ajouté au prochain commit. En revanche, on souhaite que les modifications apportées soient conservées.

On effectue alors la commande suivante : `git restore .gitignore --staged`.

```

1 # On constate que le fichier n'est plus dans la zone de staging
2 $ git status
3 On branch main
4 Changes to be committed:
5   (use "git restore --staged <file>..." to unstage)
6     new file:   README.txt
7
8 Changes not staged for commit:
9   (use "git add <file>..." to update what will be committed)
10  (use "git restore <file>..." to discard changes in working directory)
11     modified:   .gitignore
12
13 Untracked files:
14   (use "git add <file>..." to include in what will be committed)
15     doc/
16
17 # Le contenu du fichier est toujours le même
18 $ cat .gitignore
19 *.pdf
20

```

Méthode Déplacer un fichier

Lorsque l'on souhaite déplacer "proprement" un fichier ou un répertoire, il convient d'utiliser la commande `git mv` plutôt que de le déplacer via une simple commande.

Admettons que l'on souhaite déplacer le répertoire **docs** dans un répertoire **nouveauxdocs**. Il est courant d'utiliser la commande suivante : `mv docs nouveauxdocs`.

Pour autant, si l'on vérifie l'état de nos fichiers, voici ce qu'il en ressort :

```

1 $ git status
2 On branch main
3 Changes not staged for commit:
4   (use "git add/rm <file>..." to update what will be committed)
5   (use "git restore <file>..." to discard changes in working directory)
6     deleted:    docs/doc.txt.txt
7
8 Untracked files:
9   (use "git add <file>..." to include in what will be committed)
10     nouveauxdocs/
11

```

En ayant effectué cette action, Git considère que l'on a supprimé le répertoire **docs** et son contenu, et qu'un nouveau répertoire a été ajouté. Ce n'est pas ce que nous souhaitons !

La commande `git mv` permet d'effectuer cela proprement :

```

1 $ git mv docs nouveauxdocs
2
3 $ git status
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     renamed:    docs/doc.txt.txt -> nouveauxdocs/doc.txt.txt
8

```

Méthode Supprimer un fichier

Grâce à la commande `git rm`¹, il est possible de supprimer un fichier de l'espace de travail et de l'index de Git. De la même façon qu'avec `git mv`, il est préférable d'utiliser cette commande plutôt que la commande `rm`.

Il est possible de supprimer un fichier de l'index de Git sans pour autant supprimer celui-ci de l'espace de travail. Cela peut s'avérer utile si un fichier a été "commité" par erreur. Pour cela, il suffit d'utiliser l'option `--cached`.

Exemple

Dans un répertoire **download**, on a accidentellement indexé le fichier `file.txt`

On souhaite le désindexer de Git, mais on désire conserver ce fichier localement :

```
1 $ ls downloads/
2 file.txt
3
4 $ git rm downloads/file.txt --cached
5 rm 'downloads/file.txt'
6
7 $ git status
8 On branch main
9 Your branch is ahead of 'origin/main' by 2 commits.
10 (use "git push" to publish your local commits)
11
12 Changes to be committed:
13   (use "git restore --staged <file>..." to unstage)
14     deleted:    downloads/file.txt
15
16
17 $ ls downloads/
18 file.txt
19
```

Syntaxe À retenir

- La commande `git restore` permet de rétablir l'état d'origine d'un fichier.
- Pour retirer un fichier de la zone de transit, on peut ajouter l'option `--staged` à la commande `git restore` afin de rétablir son statut d'origine (modifié/non-suivi) sans perdre les modifications effectuées.
- Pour déplacer ou supprimer un fichier, il est préférable d'utiliser les commandes `git mv` ou `git rm`.

VII. Exercice : Appliquez la notion

1. Clonez le projet suivant : <https://github.com/lpe-acelys/manipuler-fichiers.git>.
2. Modifiez le fichier **README** pour y ajouter la ligne suivante : "Modification du fichier", et ajoutez-le à la zone de staging grâce à la commande `git add`.
3. Vérifiez sa présence grâce à la commande `git status`.

```
1 $ git status
2 On branch main
3 Your branch is up to date with 'origin/main'.
4
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
```

¹ <https://git-scm.com/docs/git-rm/fr>

7
8
9

modified: README.md

Question 1

[solution n°5 p.26]

Retirez ce fichier de la zone de staging sans altérer les modifications effectuées.

Question 2

[solution n°6 p.26]

Effacez désormais toutes les modifications effectuées sur ce fichier au moyen de la commande Git appropriée.

Question 3

[solution n°7 p.26]

Une erreur s'est glissée dans le nom du répertoire de documentation. Corrigez-la en déplaçant le répertoire de "documentation" vers "documentation" grâce à la commande appropriée.

Question 4

[solution n°8 p.27]

Le répertoire **downloads** a été ajouté, mais son contenu ne devrait pas être présent sur le dépôt. Supprimez le fichier qu'il contient de l'index, sans le supprimer pour autant de votre répertoire.

VIII. Ignorer des fichiers

Objectifs

- Apprendre à ignorer des fichiers
- Comprendre quels fichiers ignorer

Mise en situation

Certains fichiers présents au sein d'un répertoire ne doivent pas être versionnés, tout simplement parce qu'ils ne sont pas pertinents ou parce qu'il pourrait même s'avérer dangereux de les partager.

Grâce à un fichier appelé **.gitignore**, Git fournit une solution simple pour ignorer ces fichiers.

Méthode Ignorer des éléments

Pour permettre à Git d'ignorer certains éléments, il convient d'initialiser un nouveau fichier intitulé **.gitignore**. C'est au sein de ce fichier que devront être listés l'ensemble des éléments qui ne doivent pas être suivis.

S'il convient de l'initialiser dès le début du projet, celui-ci doit impérativement vivre au même rythme que l'évolution du projet.

Syntaxe Les syntaxes possibles

Dans ce type de fichiers, les syntaxes reconnues correspondent à des expressions régulières simplifiées couramment utilisées dans le langage shell.

Voici les différentes syntaxes à adopter :

- # permet d'ajouter un commentaire
- * permet d'ignorer tous les éléments répondant à un certain pattern, par exemple *.docx ignore tous les documents Word
- ! permet d'ajouter une exception à l'une des règles préalablement écrites, par exemple !documentation.docx
- /fichier-temporaire.tmp permet d'ignorer le fichier fichier-temporaire.tmp situé à la racine du projet

- `build/` ignore tous les éléments situés dans le répertoire **build**

Exemple Ajouter des éléments à ignorer

Voici ce que pourrait contenir un fichier **.gitignore** :

```
1 # pas de fichiers .pdf
2 *.pdf
3
4 # mais suivre documentation.pdf malgré la règle précédente
5 !documentation.pdf
6
7 # ignorer le fichier TODO à la racine du projet
8 /TODO
9
10 # ignorer tous les fichiers dans le répertoire cache
11 cache/
12
13 # ignorer tous les fichiers .txt sous le répertoire doc/
14 doc/**/*.txt
```

Conseil Quels éléments ignorer ?

Par convention, voici quelques exemples de fichiers régulièrement ignorés :

- Les répertoires de bibliothèques externes tels que les répertoires `vendor` ou `node_modules`
- Les fichiers auto-générés : répertoires de build, répertoires de cache, configuration de l'IDE
- Les fichiers dont la configuration est propre à un environnement, un utilisateur, ou contient des données sensibles (identifiants de base de données, par exemple)
- Les fichiers trop volumineux, pour lesquels d'autres solutions¹ existent

D'une manière générale, il est préférable de ne commit que les fichiers essentiels au bon fonctionnement de l'application, qui seront amenés à être modifiés et ne contenant pas de données sensibles.

Par ailleurs, de nombreux exemples de fichiers sont disponibles sur le net. GitHub met par exemple à disposition² un répertoire complet avec des configurations propres à chaque type de projet.

Complément Ignorer des éléments de façon globale

Il est possible de configurer un fichier **.gitignore** global à l'ensemble du poste de travail. Cela peut s'avérer utile lorsque l'on travaille sur beaucoup de projets dont l'architecture est similaire.

Par exemple, si l'on travaille toujours avec une IDE de type PhpStorm, il peut être intéressant d'ignorer systématiquement le répertoire **.idea** contenant la configuration de celui-ci, qu'il n'y a pas forcément lieu de partager.

Il convient dans un premier temps de créer un fichier **.gitignore** dans un répertoire commun, par exemple dans votre répertoire utilisateur.

```
1 vi ~/.gitignore
```

Ensuite, on ajoute les éléments que l'on souhaite exclure.

Enfin, on indique à Git de prendre en compte cette configuration supplémentaire :

```
1 git config --global core.excludesfile ~/.gitignore
```

1 <https://git-lfs.github.com/>

2 <https://github.com/github/gitignore>

Notez que vous avez la possibilité de quitter l'éditeur **vi** de deux façons différentes :

- :q! pour quitter sans sauvegarder le fichier
- :wq pour quitter en sauvegardant le fichier

Syntaxe À retenir

- On indique que l'on souhaite ignorer certains fichiers dans un dépôt, grâce à un fichier **.gitignore**.
- Les syntaxes reconnues sont des expressions régulières simplifiées couramment utilisées dans le langage shell.

Complément

Gérer des fichiers volumineux¹

Exemples de fichiers .gitignore mis à disposition par GitHub²

IX. Exercice : Appliquez la notion

Reprenez le dépôt `cuisiner-facile` initialisé précédemment.

Question

[solution n°9 p.27]

Initialisez un fichier **.gitignore** permettant d'ignorer les éléments suivants :

- On souhaite ignorer tous les fichiers `.txt`
- Le fichier `important.txt` ne doit pas être ignoré
- Le répertoire `/vendors` doit être ignoré
- Les fichiers `.pdf` situés dans le répertoire `uploads` doivent être ignorés

Ajoutez des commentaires permettant de comprendre rapidement à quoi correspondent les entrées de ce fichier.

Vous pouvez vérifier que votre fichier est fonctionnel en créant les fichiers adéquats.

X. Comparer des changements

Objectifs

- Apprendre à comparer des données
- Comprendre le comparatif effectué
- Identifier comment comparer des branches entre elles

Mise en situation

Avant d'effectuer un commit (ou même en cours de développement), il est important de vérifier et valider quels éléments ont été modifiés. Si la commande `git status` est déjà un bon indicateur, il est primordial de comparer dans le détail ce qui a été modifié entre deux versions.

¹ <https://git-lfs.github.com/>

² <https://github.com/github/gitignore>

Méthode Comparer les changements

Il est possible d'effectuer cette comparaison grâce à la commande `git diff`¹. Cette commande permet d'effectuer des comparaisons entre plusieurs sources de données Git. Par défaut, cette commande permet de comparer l'état des fichiers depuis le dernier commit.

Néanmoins, il est possible de comparer d'autres éléments entre eux, tels que des commits ou des branches.

Exemple

Supposons que nous disposions d'un fichier **README** dont le contenu initial lors du dernier commit était le suivant :

```
1 Initialisation du fichier README
2
3 Une information
```

Nous l'avons ensuite modifié de telle sorte, en y ajoutant une nouvelle ligne :

```
1 Initialisation du fichier README
2
3 Une information modifiée
4
5 Une nouvelle information
6
```

Si l'on exécute la commande `git diff`, voici ce qu'il en ressortira :

```
1 $ git diff
2 diff --git a/README.txt b/README.txt
3 index 7778e04..9bee702 100644
4 --- a/README.txt
5 +++ b/README.txt
6 @@ -1,3 +1,5 @@
7  Initialisation du fichier README
8
9 -Une information
10 +Une information modifiée
11 +
12 +Une nouvelle information
13
```

Méthode Comment lire ce résultat

```
1 diff --git a/README.txt b/README.txt
```

Cette ligne indique les fichiers ayant été comparés, ici on constate qu'il s'agit du fichier **README.txt**.

```
1 index 7778e04..9bee702 100644
```

Ici, il s'agit des métadonnées internes de Git, représentant les identifiants des objets.

```
1 --- a/README.txt
2 +++ b/README.txt
```

Ces lignes représentent les marqueurs de changements, il s'agit d'une légende. Les `---` représentent les informations retirées, `+++` les informations ajoutées.

Ensuite, l'affichage ne représente que les éléments modifiés du fichier.

```
1 @@ -1,3 +1,5 @@
```

¹ <https://git-scm.com/docs/git-diff>

Ici, on constate qu'une ligne a été retirée à la ligne 3 (il s'agit de la ligne modifiée), tandis qu'une nouvelle ligne a été ajoutée à la ligne 5.

Attention

Les données ayant déjà été ajoutées à la zone de staging grâce à `git add` ne seront par défaut pas présentes dans le *diff*, **veillez donc à utiliser cette commande avant d'effectuer un `git add`**.

Si vous souhaitez tout de même visualiser ces modifications, il conviendra d'utiliser l'option `--cached`.

Complément Ajouter un peu de granularité

Si le *diff* n'est pas assez précis, il est possible d'ajouter l'option `--color-words`.

Le détail des éléments modifié se fera alors par mots, plutôt que par ligne.

```
$ git diff --color-words
diff --git a/README.md b/README.md
index 38731d3..682d42f 100644
--- a/README.md
+++ b/README.md
@@ -1,3 @@
Initialisation du fichier !

Ajout d'une nouvelle ligne

$ git diff
diff --git a/README.md b/README.md
index 38731d3..682d42f 100644
--- a/README.md
+++ b/README.md
@@ -1,3 @@
-Initialisation du fichier
+Initialisation du fichier !
+
+Ajout d'une nouvelle ligne
```

Syntaxe À retenir

- On compare des modifications grâce à la commande `git diff`.
- L'option `--color-words` permet de visualiser le détail de changements par mots plutôt que par ligne.
- Par défaut, les données comparées sont les données modifiées depuis le dernier commit.
- Si des modifications ont déjà été ajoutées à la zone de staging, elles ne seront pas visibles par défaut, il faudra utiliser l'option `--cached`.

XI. Exercice : Appliquez la notion

Retournez dans le répertoire `manipuler-fichier` précédemment cloné, et exécutez la commande `git reset --hard origin/master` afin de remettre le répertoire dans son état d'origine.

Si vous n'avez plus ce répertoire, clonez-le à partir de ce lien : <https://github.com/lpe-acelys/manipuler-fichiers.git>.

Question 1

[solution n°10 p.27]

La commande `git diff` affiche par défaut le différentiel de l'ensemble des modifications apportées n'ayant pas été ajoutées à la commande de staging.

À l'aide de la documentation¹ ou de ce que vous avez pu voir précédemment, identifiez comment vous pourriez effectuer un différentiel sur un seul fichier.

Question 2

[solution n°11 p.27]

Modifiez les fichiers suivants :

- `README.md` : "Ajout d'une nouvelle ligne"
- `documentation/documentation.md` : "Mise à jour de la documentation"

Affichez le différentiel du fichier `README.md` uniquement.

XII. Valider une modification**Objectifs**

- Identifier quand valider les modifications
- Savoir comment valider les modifications

Mise en situation

Il s'agit de la dernière étape du processus. Les modifications sur les différents fichiers ont été effectuées, on sait ce que l'on souhaite conserver, les vérifications nécessaires ont été effectuées : il est temps d'enregistrer notre travail.

Rappel Enregistrer son travail régulièrement

Nous l'avons déjà évoqué, mais il est primordial d'enregistrer son travail régulièrement. Toute modification apportée sur un projet devrait être sauvegardée dès qu'elle a été testée et que le projet se trouve dans un état fonctionnel.

Qu'il s'agisse d'une nouvelle fonctionnalité ou de la correction d'un bug, ces enregistrements d'états doivent être les plus réguliers possibles. Cela permettra aux collaborateurs du projet d'identifier facilement dans le temps quelles modifications ont été apportées au projet en visualisant l'historique de celui-ci, quels changements ont été faits et à quel moment.

Méthode Enregistrer ses modifications

Afin d'enregistrer ses modifications (après avoir vérifié l'état des modifications grâce à la commande `git status`), il suffit de lancer la commande `git commit`².

En lançant cette commande, on constate l'interface ci-dessous. C'est à ce moment qu'il faut saisir un nouveau message indiquant quelles modifications ont été effectuées.

Cette interface permet également de visualiser quels changements seront commités.

```
1 fix: déplacement du répertoire doc -> docs
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # Committer: User
6 #
7 # On branch main
```

1 <https://git-scm.com/docs/git-diff#Documentation/git-diff.txt---color-words!tregexgt>

2 <https://git-scm.com/docs/git-commit>

```

8 # Changes to be committed:
9 #       renamed:    doc/doc.txt.txt -> docs/doc.txt.txt
10 #
11 ~
12 ~
13

```

Pour effectuer un commit rapidement, il est possible d'utiliser l'option `-m` et d'indiquer le message que l'on souhaite commit directement.

Ainsi, on aurait pu écrire `git commit -m "fix: déplacement du répertoire doc -> docs"`.

Méthode Vérifier que son commit a bien été pris en compte

Une fois le commit effectué, on peut vérifier que celui-ci a bien été pris en compte grâce à la commande `git log`.

```

1 $ git log
2 commit e4bf18c41c745ab2aed6e7de73acb5bb6f8f6813 (HEAD -> main)
3 Author: User
4 Date:   Thu May 7 18:02:03 2020 +0200
5
6     fix: déplacement du répertoire doc -> docs
7

```

Si l'on exécute la commande `git status`, on constatera que les fichiers non ajoutés sont toujours présents, ou que notre espace de travail est désormais "propre", si tous les fichiers ont été ajoutés.

```

1 $ git status
2 On branch main
3 nothing to commit, working tree clean

```

Conseil L'importance d'un message clair

Lorsque l'on effectue un commit, que l'on travaille seul ou en équipe, il est important de conserver des messages de commits clairs et précis. Un projet peut vivre de nombreuses années, son historique le suivra, il est donc important de pouvoir conserver quelque chose de précis et que l'on peut parcourir facilement.

La modification apportée était-elle un correctif ? Une nouvelle fonctionnalité ? Qu'est-ce qui a été fait ? Un ticket Trello/Jira ou autre y était-il associé et permettrait-il de s'y référer en cas de besoin ? Ces éléments sont importants pour le futur.

De nombreuses conventions¹ de nommage existent, chaque équipe dispose souvent de ses propres conventions. Qu'importe la forme, le principal étant de les appliquer avec constance et rigueur.

Syntaxe À retenir

- On enregistre ses modifications avec la commande `git commit`.
- Il est important d'enregistrer son travail régulièrement.
- Un message clair et des conventions respectées par l'équipe permettent de faciliter la lecture et la compréhension de l'historique dans le temps.

¹ <https://www.conventionalcommits.org/fr/v1.0.0-beta.3/>

XIII. Exercice : Appliquez la notion

Retournez dans le répertoire `manipuler-fichier` précédemment cloné et exécutez la commande `git reset --hard origin/master` afin de remettre le répertoire dans son état d'origine.

Si vous n'avez plus ce répertoire, clonez-le à partir de ce lien : <https://github.com/lpe-acelys/manipuler-fichiers.git>.

Question 1

[solution n°12 p.28]

Nous l'avons remarqué précédemment, le répertoire `documentation` est mal nommé, une faute de frappe a été oubliée et il a été nommé "`documentatation`".

Déplacez ce répertoire à l'aide de la commande `git mv` et commitez vos modifications après avoir vérifié le contenu de votre zone de staging grâce à la commande `git status`.

Les commits ayant été effectués précédemment sont nommés de la sorte :

```
1 MISC: Add downloads directory
2
3 DOC: Init documentation file
4
5 DOC: Create README file
6
```

Pour conserver une cohérence, vous nommerez donc votre commit "`MISC: Move documentation directory`".

Question 2

[solution n°13 p.28]

Modifiez le fichier `README.md` en y ajoutant une nouvelle ligne "*Ajout d'une nouvelle ligne*".

Effectuez les manipulations de vérifications et d'ajouts nécessaires au bon processus de validation des modifications : vérification des modifications apportées, état de la zone de staging, ajout à la zone de staging.

Commitez les modifications apportées avec le message suivant "`DOC : Update README file`".

XIV. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°14 p.29]

Exercice

Lorsque l'on démarre un nouveau projet, quelle(s) commande(s) peut-on utiliser pour initialiser un nouveau dépôt ?

- ☐ `git pull`
- ☐ `git init`
- ☐ `git clone`
- ☐ `git new`

Exercice

Grâce à quelle commande est-il possible de vérifier l'état de l'espace de travail ?

- ☐ `git show`
- ☐ `git status`
- ☐ `git diff`

Exercice

Au sein de quel répertoire l'ensemble des données du dépôt sont-elles synchronisées ?

Exercice

Soit l'état du dépôt suivant :

```
1 On branch main
2 Changes to be committed:
3   new file:   README.txt
4
5 Changes not staged for commit:
6   modified:   .gitignore
7
8 Untracked files:
9   doc/
10
```

Quelle commande faudrait-il utiliser pour ajouter à notre prochain commit le fichier **.gitignore** ?

- ☐ git add .gitignore
- ☐ git restore .gitignore
- ☐ git commit .gitignore

Exercice

Le fichier **README** a été modifié, mais n'a pas encore été ajouté à la zone de staging. Comment peut-on annuler les modifications effectuées ?

- ☐ git rm README
- ☐ git ignore README
- ☐ git restore README

Exercice

Lorsque l'on souhaite déplacer proprement un fichier, il est préférable d'utiliser...

- ☐ mv file_to_move
- ☐ git mv file_to_move

Exercice

Dans quel fichier les éléments ne devant pas être commités doivent-ils être ajoutés ?

Exercice

Soit le fichier d'exclusion suivant :

```
1 *.txt
2
3 !important.txt
4
5 vendors/
6
7 uploads/**/*.pdf
```

Indiquez quels fichiers seront ignorés par Git parmi la liste suivante.

- ☐ document.txt
- ☐ important.txt
- ☐ uploads/documentation.pdf
- ☐ vendors/vendor.json
- ☐ uploads/documentation.txt

Exercice

Afin de comparer les modifications apportées entre deux versions, quelle commande peut-on utiliser ?

- ☐ git diff
- ☐ git status
- ☐ git compare

Exercice

Soit l'état des fichiers suivants :

```
1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     new file:   README.txt
8
9 Changes not staged for commit:
10    (use "git add <file>..." to update what will be committed)
11    (use "git restore <file>..." to discard changes in working directory)
12     modified:   .gitignore
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16     doc/
17
```

Indiquez quel(s) fichier(s) sera/seront ajouté(s) lors du prochain commit.

- ☐ README.txt
- ☐ .gitignore
- ☐ doc/

B. Exercice : Défi

Vous rejoignez une nouvelle équipe dans le but de travailler sur la refonte d'un site Internet pour la société Laure & Mipsum. Il s'agit des prémices du projet, le dépôt vient d'être initialisé.

Ce projet est accessible à cette adresse : <https://github.com/lpe-acelys/refonte-site-web>.

Vous découvrez vos premières tâches.

Question 1

[solution n°15 p.31]

Vous devez dans un premier temps initialiser le projet sur votre environnement local.

Cette refonte n'étant pas votre premier projet, vous clonerez le dépôt en le nommant "refonte-laure-et-mipsum".

Question 2

[solution n°16 p.31]

Vous êtes chargé d'initialiser un fichier **.gitignore**.

On considère que les fichiers `.txt`, `docx` et `pdf` seront systématiquement ignorés, tout comme les répertoires `uploads` et `vendors`.

Le répertoire `uploads` contient un fichier `README.md` qu'il conviendra de conserver.

Vous commiterez les modifications apportées dans un commit intitulé "MISC : Init `.gitignore`".

Question 3

[solution n°17 p.31]

Le dossier de documentation n'a pas été nommé de façon suffisamment explicite.

Vous êtes chargé de le renommer en "documentation" et de commiter ces modifications dans un commit intitulé "DOC: Rename documentation direction".

Solutions des exercices

p. 6 Solution n°1

Voici ce que vous avez dû obtenir. **Conservez ce dépôt.**

```
1 user@user MINGW64 /e/workspace/git-bases
2 $ mkdir cuisiner-facile
3
4 user@user MINGW64 /e/workspace/git-bases
5 $ cd cuisiner-facile/
6
7 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile
8 $ git init
9 Initialized empty Git repository in E:/workspace/git-bases/cuisiner-facile/.git/
10
11
```

p. 6 Solution n°2

```
1 user@user MINGW64 /e/workspace/git-bases
2 $ git clone https://github.com/thedaviddias/Front-End-Performance-Checklist.git
3 Cloning into 'Front-End-Performance-Checklist'...
4 remote: Enumerating objects: 19, done.
5 remote: Counting objects: 100% (19/19), done.
6 remote: Compressing objects: 100% (17/17), done.
7 remote: Total 526 (delta 6), reused 7 (delta 2), pack-reused 507
8 Receiving objects: 100% (526/526), 315.37 KiB | 1.11 MiB/s, done.
9 Resolving deltas: 100% (300/300), done.
10
11 user@user MINGW64 /e/workspace/git-bases
12 $ ls -la
13 total 16
14 drwxr-xr-x 1 user 1049089 0 mai 11 09:54 ./
15 drwxr-xr-x 1 user 1049089 0 mai 7 10:24 ../
16 drwxr-xr-x 1 user 1049089 0 mai 11 09:54 Front-End-Performance-Checklist/
17
```

p. 10 Solution n°3

```
1 $ git status
2 On branch main
3
4 No commits yet
5
6 Untracked files:
7   (use "git add <file>..." to include in what will be committed)
8     README
9
10 nothing added to commit but untracked files present (use "git add" to track)
11
```

p. 10 Solution n°4

```

1 $ git add README
2
3 $ git status
4 On branch main
5
6 No commits yet
7
8 Changes to be committed:
9   (use "git rm --cached <file>..." to unstage)
10    new file:   README
11
12

```

p. 14 Solution n°5

```

1 $ git restore README.md --staged
2
3 $ git status
4 On branch main
5 Your branch is up to date with 'origin/main'.
6
7 Changes not staged for commit:
8   (use "git add <file>..." to update what will be committed)
9   (use "git restore <file>..." to discard changes in working directory)
10    modified:   README.md
11
12

```

p. 14 Solution n°6

```

1 $ git restore README.md
2
3 $ git status
4 On branch main
5 Your branch is up to date with 'origin/main'.
6
7 nothing to commit, working tree clean
8

```

p. 14 Solution n°7

```

1 git mv documentation/ documentation
2
3 $ git status
4 On branch main
5 Your branch is up to date with 'origin/main'.
6
7 Changes to be committed:
8   (use "git restore --staged <file>..." to unstage)
9    renamed:    documentation/documentation.md -> documentation/documentation.md
10

```

p. 14 Solution n°8

```
1 git rm downloads/file.txt --cached
```

p. 16 Solution n°9

```
1 # On ignore les fichiers .txt
2 *.txt
3
4 # On conserve important.txt
5 !important.txt
6
7 # On ignore le répertoire /vendors
8 vendors/
9
10 # On ignore les fichiers pdf du répertoire uploads
11 uploads/**/*.pdf
12
13 $ ls -R -l
14 .:
15 ignore.txt
16 important.txt
17 README
18 uploads/
19 vendors/
20
21 ./uploads:
22 file.docx
23 file.pdf
24
25 ./vendors:
26 vendor
27
28 $ git status
29 On branch main
30 Untracked files:
31   (use "git add <file>..." to include in what will be committed)
32       .gitignore
33       important.txt
34       uploads/
35
```

p. 19 Solution n°10

Il suffit d'exécuter la commande `git diff nom_du_fichier`.

p. 19 Solution n°11

```

1 $ git diff README.md
2 diff --git a/README.md b/README.md
3 index 38731d3..1e078fc 100644
4 --- a/README.md
5 +++ b/README.md
6 @@ -1,3 @@
7  Initialisation du fichier
8  +
9  +Ajout d'une nouvelle ligne
10

```

p. 21 Solution n°12

```

1 $ git mv documentation documentation
2
3 $ git status
4 On branch main
5 Your branch is up to date with 'origin/main'.
6
7 Changes to be committed:
8   (use "git restore --staged <file>..." to unstage)
9     renamed:    documentation/documentation.md -> documentation/documentation.md
10
11 $ git commit -m "MISC: Move documentation directory"
12

```

p. 21 Solution n°13

On vérifie l'état de nos fichiers :

```

1 $ git status
2 On branch main
3 Your branch is ahead of 'origin/main' by 1 commit.
4   (use "git push" to publish your local commits)
5
6 Changes not staged for commit:
7   (use "git add <file>..." to update what will be committed)
8   (use "git restore <file>..." to discard changes in working directory)
9     modified:   README.md
10
11 no changes added to commit (use "git add" and/or "git commit -a")
12

```

On vérifie également quelles modifications ont été apportées :

```

1 $ git diff
2 diff --git a/README.md b/README.md
3 index 38731d3..1e078fc 100644
4 --- a/README.md
5 +++ b/README.md
6 @@ -1,3 @@
7  Initialisation du fichier
8  +
9  +Ajout d'une nouvelle ligne
10
11

```

On ajoute le fichier :

```
1 $ git add README.md
2
```

On commit les modifications :

```
1 $ git commit -m "DOC : Update README file"
2
```

Exercice p. 21 Solution n°14

Exercice

Lorsque l'on démarre un nouveau projet, quelle(s) commande(s) peut-on utiliser pour initialiser un nouveau dépôt ?

- ☐ git pull
- ☒ git init
- ☐ git clone
- ☐ git new

Exercice

Grâce à quelle commande est-il possible de vérifier l'état de l'espace de travail ?

- ☐ git show
- ☒ git status
- ☐ git diff

Exercice

Au sein de quel répertoire l'ensemble des données du dépôt sont-elles synchronisées ?

.git

Exercice

Soit l'état du dépôt suivant :

```
1 On branch main
2 Changes to be committed:
3   new file:   README.txt
4
5 Changes not staged for commit:
6   modified:   .gitignore
7
8 Untracked files:
9   doc/
10
```

Quelle commande faudrait-il utiliser pour ajouter à notre prochain commit le fichier **.gitignore** ?

- ☒ git add .gitignore
- ☐ git restore .gitignore
- ☐ git commit .gitignore

Exercice


Le fichier **README** a été modifié, mais n'a pas encore été ajouté à la zone de staging. Comment peut-on annuler les modifications effectuées ?

- ☐ git rm README
- ☐ git ignore README
- ☒ git restore README

Exercice

Lorsque l'on souhaite déplacer proprement un fichier, il est préférable d'utiliser...

- ☐ mv file_to_move
- ☒ git mv file_to_move

 Il est préférable d'utiliser `git mv`. En utilisant simplement `mv`, Git considère que le fichier est supprimé puis recréé, tandis qu'il le considère véritablement comme déplacé grâce à la commande `git mv`.

Exercice

Dans quel fichier les éléments ne devant pas être commités doivent-ils être ajoutés ?

.gitignore

Exercice

Soit le fichier d'exclusion suivant :

```
1 *.txt
2
3 !important.txt
4
5 vendors/
6
7 uploads/**/*.pdf
```

Indiquez quels fichiers seront ignorés par Git parmi la liste suivante.

- ☒ document.txt
- ☐ important.txt
- ☒ uploads/documentation.pdf
- ☒ vendors/vendor.json
- ☒ uploads/documentation.txt

Les fichiers `txt` sont tous exclus à l'exception de `important.txt`.

Exercice

Afin de comparer les modifications apportées entre deux versions, quelle commande peut-on utiliser ?

- ☒ git diff
- ☐ git status
- ☐ git compare

Exercice

Soit l'état des fichiers suivants :

```

1 user@user MINGW64 /e/workspace/git-bases/cuisiner-facile (main)
2 $ git status
3
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7     new file:   README.txt
8
9 Changes not staged for commit:
10    (use "git add <file>..." to update what will be committed)
11    (use "git restore <file>..." to discard changes in working directory)
12    modified:   .gitignore
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16     doc/
17

```

Indiquez quel(s) fichier(s) sera/seront ajouté(s) lors du prochain commit.

- ☒ README.txt
- ☐ .gitignore
- ☐ doc/

p. 23 Solution n°15

Voici la commande que vous avez dû effectuer :

```
1 git clone https://github.com/lpe-acelys/refonte-site-web.git refonte-laure-et-mipsum
```

p. 24 Solution n°16

Voici la commande que vous avez dû effectuer :

```

1 *.txt
2 *.docx
3 *.pdf
4
5 uploads/
6 vendors/
7
8 !uploads/README.md
9
1 git add .gitignore
2 git commit -m "MISC: Init .gitignore"

```

p. 24 Solution n°17

```

1 git mv doc/ documentation
2 git commit -m "DOC: Rename documentation direction"

```