

CEFii



www.cefii.fr



git

GitHub



Git & GitHub



Introduction

• Introduction

Dans ce tutoriel, vous découvrirez ce qu'est le **versioning** et comment le mettre en œuvre avec l'outil **Git**.

Nous verrons également comment utiliser la plateforme **GitHub** pour partager nos projets.

• Objectifs

- Principe du versioning
- Installation de Git
- Configuration
- Création d'un nouveau projet
- Mise en pratique du versioning
 - Commit
 - Gestion des branches
 - Retour en arrière
- Dépôt distant – GitHub
- Cloner un projet
- Interfaces graphiques

Principe du versioning

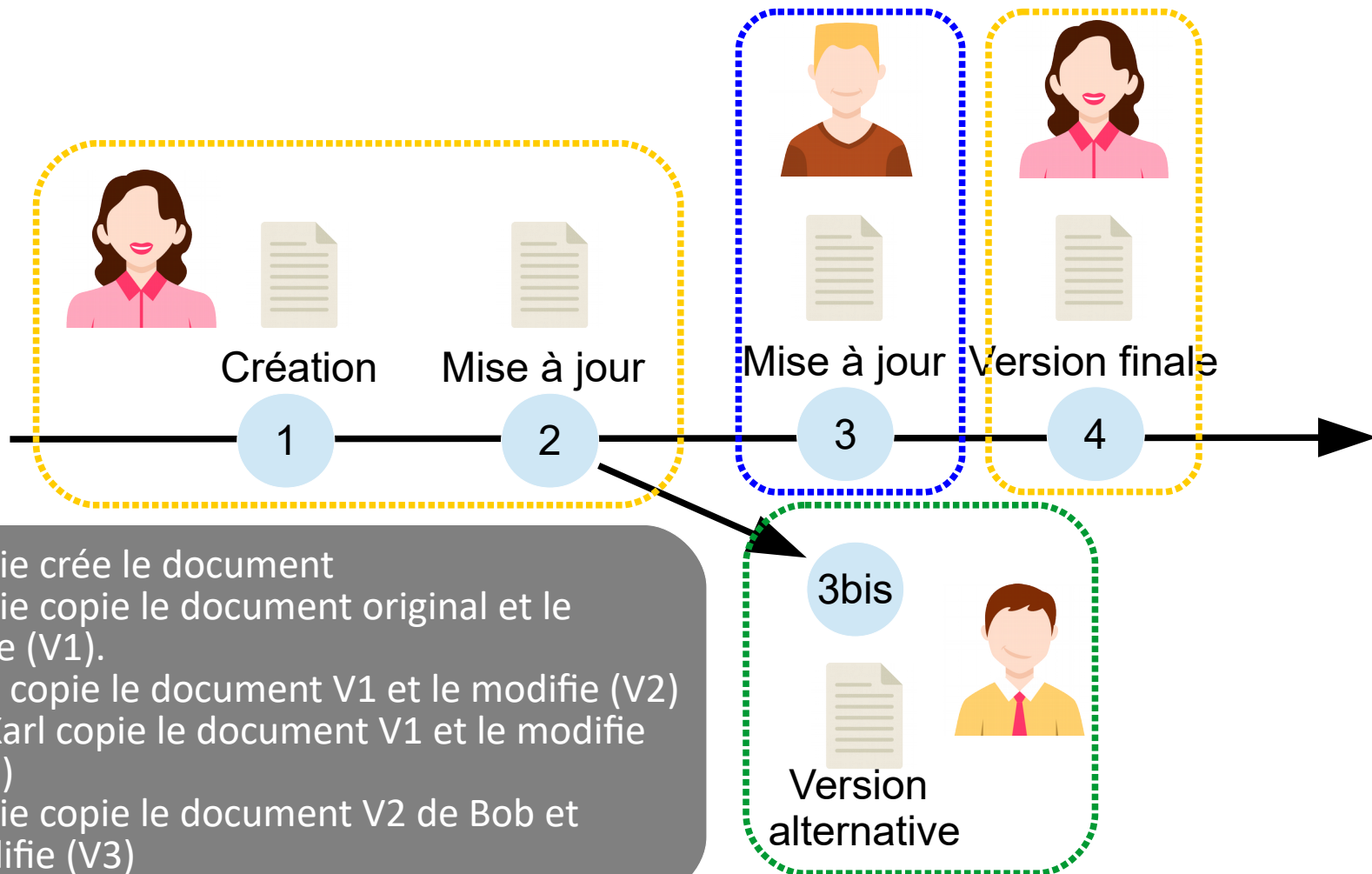


1

Le versioning

• Le versioning

Problématique - exemple de cycle de vie (*workflow*) d'un fichier :





1

Le versioning

• Le versioning

Problématique

Pour suivre ce qui se passe, il faut disposer des informations suivantes :

- Qui : quel acteur intervient sur le fichier ?
- Quand : à quel moment cet acteur est-il intervenu ?
- Quoi : ce qui a été modifié dans le code
- Pourquoi : ce qui a motivé ces modifications

Solution

Pour gérer les nombreuses problématiques qui se posent tout au long du cycle de vie d'un fichier, voire d'une application, on peut recourir à des logiciels de gestion de versions (ou VCS pour *Version Control System*) qui permettent de :

- **stocker** un ensemble de fichiers en conservant la chronologie de toutes les modifications qui ont été effectuées dessus.
- **retrouver** les différentes versions d'un lot de fichiers connexes.



1

Le versioning



• Le versioning

Objectifs d'un gestionnaire de versions

1. Conserver un **historique** des modifications, pour revenir sur une étape précédente du développement (si besoin).
2. Travailler en **équipe** en gérant la problématique des **conflits de fusion** qui peuvent apparaître lorsque plusieurs développeurs travaillent sur le même code.
3. Supporter des **branches** de développement, ce qui permet de décliner le projet en sous-projets.
4. On pourra ainsi conserver un projet principal et tester des projets dérivés. Si les modifications apportées sur une branche sont bonnes, on les ramène sur le projet principal, sinon on abandonne la branche.

Il existe des logiciels et services de **gestion de versions décentralisée** (distribué) (ou DVCS pour *Distributed Version Control System*).

Git et **Mercurial** en sont deux exemples disponibles sur la plupart des systèmes Unix et Windows.



1

Le versioning

• Le versioning

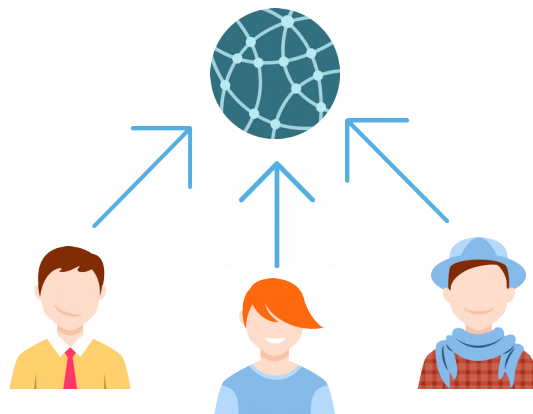
Problème du VCS : travail sur un serveur centralisé

- Tout est stocké sur un serveur externe. En cas de problème, perte des données
- Lenteur d'accès dans les transferts de données

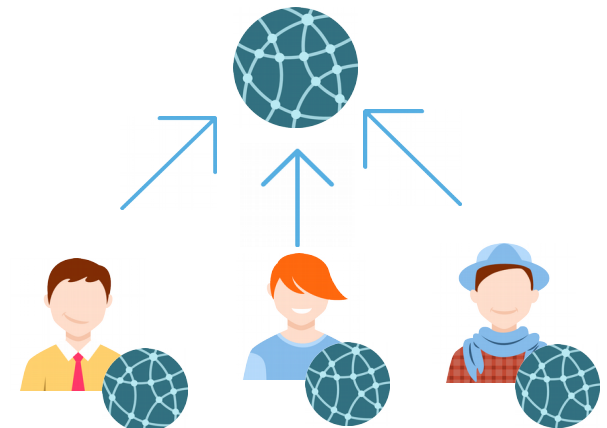
Solution de Git : travailler de manière **décentralisée**.

Chaque utilisateur a un **historique** du projet en local, ce qui fait qu'il travaille en 2 temps :

- 1/ Développement du projet en local (ce qui va plus vite)
- 2/ Remontée du code sur le serveur.



VCS classique



Git

Installation de Git



2

Installation

• Installation de Git

Pour les OS Windows et Mac, aller sur le site <https://git-scm.com/> et cliquez sur le lien de téléchargement, puis suivez les étapes d'installation pas à pas.

Pour la suite de notre travail, nous allons devoir utiliser un **terminal** (on parle parfois de console également) afin de taper des instructions en ligne de commande.



Vous pouvez utiliser le terminal natif de votre OS ou bien en télécharger un autre.

Nous vous proposons d'utiliser GitBash sous Windows, téléchargeable à l'adresse <https://git-scm.com/downloads>





2

Installation

• Installation de Git

Le travail en ligne de commande est différent selon qu'on soit en DOS (sous Window) ou Linux (sous Mac OS, Ubuntu ou encore avec Git Bash par exemple). Voici quelques notions de base.

Le terminal est une interface qui permet d'exécuter des actions plus rapidement ou tout simplement non accessibles depuis le système classique.

Après avoir écrit et lancé une commande en appuyant sur Enter, celle-ci est exécutée ; il sera ensuite possible de lancer une nouvelle commande dès que le terminal nous laissera la main (présence du curseur) .

Il est possible de rappeler une commande précédente en appuyant sur les flèches haut / bas du clavier.

Quelques commandes courantes :

- **cd** *nomDuDossier* permet d'entrer dans le dossier « nomDuDossier »
- **cd** ../ permet de remonter d'un cran dans l'arborescence
- **mkdir** *nomDuDossier* permet de créer un dossier « nomDuDossier »
- **help** accéder à la liste de commandes disponibles
- **Ctrl + C** interrompt une exécution en cours
- **exit** ferme le terminal



2

Installation

- **Installation de Git**

Une fois l'installation finalisée, vérifiez que Git est bien installé en allant en console et en tapant l'instruction suivante:

```
$ git --version
```

(ici le \$ indique le début de la ligne de commande et n'est pas à écrire)

Le résultat s'affiche à suivre dans la console :

```
agnes@DESKTOP-KQAH1DD MINGW64 ~  
$ git --version  
git version 2.13.2.windows.1
```

Petite variante pour l'installation sous Linux, qui se fait directement en console :

```
$ sudo apt-get install git
```

Configuration



3

Configuration

• Configuration

Dans un premier temps, nous allons configurer un certain nombre de paramètres dans Git afin de décliner notre identité.

Ainsi n'aura-t-on pas besoin de refaire cette opération à chaque intervention par la suite.

Définir l'identité de l'utilisateur

2 types de config sont possibles :

- Globale
- Propre à un dépôt

Il y a 2 variables à configurer pour identifier l'utilisateur : **user.email** et **user.name**

Pour cela, on tape la commande :

```
$ git config --global user.name "My name"  
$ git config --global user.email "mymail@domain.com"
```



3

Configuration

• Configuration

Astuce : Pour voir quelles sont les options de configuration, tapez la commande

```
$ git help config
```

Le résultat étant une page HTML, elle s'ouvrira dans votre navigateur :

git-config(1) Manual Page

NAME

git-config - Get and set repository or global options

SYNOPSIS

```
git config [<file-option>] [type] [--show-origin] [-z | --null] name [value [value_regex]]
git config [<file-option>] [type] --add name value
git config [<file-option>] [type] --replace-all name value [value_regex]
git config [<file-option>] [type] [--show-origin] [-z | --null] --get name [value_regex]
git config [<file-option>] [type] [--show-origin] [-z | --null] --get-all name [value_regex]
git config [<file-option>] [type] [--show-origin] [-z | --null] [--name-only] --get-regexp name_regex [value_regex]
```



3

Configuration

• Configuration

Il est possible de forcer la coloration syntaxique en modifiant la valeur par défaut du paramètre *color.ui* avec la commande :

```
$ git config --global color.ui true
```

Pour vérifier les paramètres de la configuration:

```
$ git config --list
```

```
$ $ git config --list
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.required=true
filter.lfs.process=git-lfs filter-process
credential.helper=manager
user.email=a.rahard@cefii.fr
user.name=Agnès CEFii
color.ui=true
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
```

Création d'un projet



4

Nouveau projet



• Nouveau projet

Pour créer un nouveau projet avec Git (on parle également de dépôt Git), cela se passe en 3 temps :

1/ Dans la console, nous allons créer un dossier pour le projet avec la commande suivante:

```
$ mkdir projet
```

A noter : cette commande produit la même chose que si l'on procède à l'aide de l'explorateur Windows ou du Finder sous Mac.

2/ Allez dans ce dossier avec l'instruction :

```
$ cd projet
```

3/ Initialisez Git :

```
$ git init
```



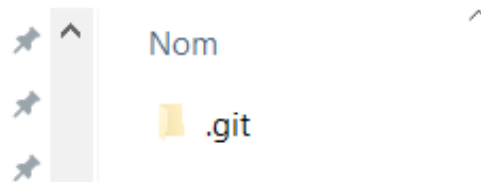
4

Nouveau projet

- **Nouveau projet**

Ceci crée un fichier caché .git dans le répertoire courant

Ce PC > Disque local (C:) > wamp64 > www > git > projet



Pour vérifier ce qui est en place dans cet espace de travail, tapez l'instruction :

```
$ git status
```

Cette instruction fait un état de lieu par rapport au contenu du dépôt et indique le statut des fichiers en place par rapport à Git (déjà traités, en attente, etc.).

Ici, rien à signaler puisque notre répertoire est encore vide...

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```



4

Nouveau projet



index.html

• Nouveau projet

Nous allons maintenant ajouter un fichier dans notre dossier *projet* en créant un script ***index.html*** contenant les instructions suivantes:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1>Titre de ma page</h1>
  <p>Paragraphe 1</p>
  <p>Paragraphe 2</p>
  <p>Paragraphe 3</p>
</body>
</html>
```

Si on lance à nouveau l'instruction

```
$ git status
```

Git va détecter l'apparition d'un nouvel élément dans son espace de travail et le signaler ainsi :

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)
```



4

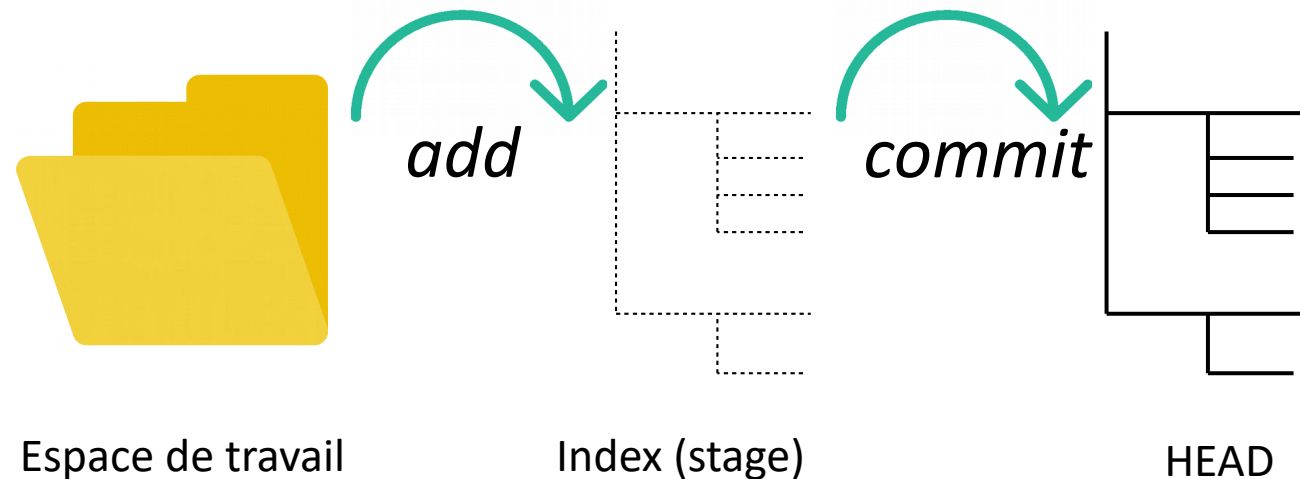
Nouveau projet

• Nouveau projet

Pour bien comprendre comment fonctionne Git, il faut voir notre espace de travail comme un dépôt local composé de trois "arbres" gérés par Git.

1. L'espace de travail (le dossier) qui contient réellement les fichiers.
2. Un index (stage) qui joue un rôle d'espace de transit pour les fichiers
3. Le HEAD qui pointe vers la dernière validation faite.

Lorsqu'on va vouloir sauvegarder notre travail (on parle de « commiter ») ceci se passe en 2 étapes :





4

Nouveau projet

- **Nouveau projet**

Lorsqu'on intervient sur les fichiers (ajout/modif/suppression), Git perçoit qu'il y a eu un changement et positionne les fichiers concernés dans un état « entre deux » sorte de sas avant la sauvegarde.





4

Nouveau projet

• Nouveau projet

Il apparaît sur le dernier *git status* que Git a relevé un élément non encore traité (Untracked files) : c'est notre fichier `index.html`

A suivre, Git nous indique l'instruction qui va permettre l'ajout de ce fichier avec Git.

Cette opération se passe donc en 2 temps comme expliqué précédemment :

1. l'ajout du fichier dans une zone de traitement intermédiaire, sorte de sas entre les éléments non traités et ceux « commités », c'est à dire dont la version est enregistrée avec Git.

Ceci se fait avec la commande ***add*** :

```
$ git add index.html
```

2. Le traitement de ces éléments stockés dans l'espace intermédiaire pour les enregistrer avec Git, en précisant avec un commentaire (texte libre) ce qui est fait :

```
$ git commit -m "initialisation projet"
```



4

Nouveau projet

- **Nouveau projet**

Résultat en console :

```
1 agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git add index.html

2 agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git commit -m "initialisation du projet"
[master (root-commit) 93a923a] initialisation du projet
1 file changed, 13 insertions(+)
create mode 100644 index.html

3 agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git status
On branch master
nothing to commit, working tree clean
```

1. Ajout de l'index dans l'espace intermédiaire (stage)
2. Commit des éléments dans cet espace intermédiaire
3. Etat des lieux : plus aucun élément n'est en attente de traitement (nothing to commit)

Commit



5

Modification



index.html

- **Modification**

Nous allons maintenant modifier notre script afin de pouvoir en sauvegarder une nouvelle version :

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Mon site web</title>
</head>
<body>
  <h1>Bienvenue!</h1>
  <div>Article 1</div>
  <div>Article 2</div>
</body>
</html>
```

A partir de quoi nous allons réitérer les opérations vues précédemment :

1. ajout de l'élément à sauvegarder
2. Commit de l'espace de travail intermédiaire



5

Modification

• Modification

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git add index.html

agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git commit -m "modification du contenu de la page d'accueil"
[master 5522cb3] modification du contenu de la page d'accueil
1 file changed, 4 insertions(+), 5 deletions(-)
```

Pour avoir une vue précise de ce qui s'est passé depuis le début du projet, on utilise la commande

```
$ git log
```



5

Modification

• Modification

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git log
commit 5522cb3cfb7d9cb73c8f1f84f4c33537f5e38710 (HEAD -> master)
Author: Agnès CEFii <a.rahard@cefii.fr>
Date:   Wed Mar 6 16:37:40 2019 +0100

    modification du contenu de la page d'accueil

commit 93a923afe0a7c342ee958850383fd2142f8fdb9e
Author: Agnès CEFii <a.rahard@cefii.fr>
Date:   Wed Mar 6 16:25:53 2019 +0100

    initialisation du projet
```

La commande affiche tous les commits qui ont été faits depuis le début du projet (depuis l'initialisation de Git dans l'espace de travail).

Ici, on voit que l'on a fait 2 commits avec pour chacun

- Une signature (chaîne alpha-numérique) qui permet de l'identifier de manière unique
- Son auteur
- La date du commit
- Le texte descriptif



5

Modification

• Modification

On peut utiliser des sélecteurs génériques comme par exemple ici l'astérisque « * » qui indique que l'on sélectionne tous les fichiers quel que soit leur nom, ayant comme extension « html »


```
$ git add *.html
```

On peut aussi utiliser l'instruction **--all** afin de sélectionner tous les fichiers (ajouts et suppressions inclus).

```
$ git add --all
```

Enfin il est possible de faire les 2 étapes en une seule commande :

```
$ git commit -a -m « deuxième commit »
```



```
$ git commit -a -m "deuxième commit"  
[master f400b45] deuxième commit  
1 file changed, 2 insertions(+), 1 deletion(-)
```

Les branches



6

Branches

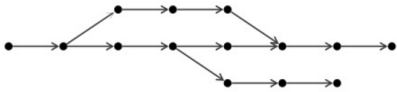
• Les branches

L'organisation en branches permet :

- d'avoir un historique non linéaire
- de travailler sur une fonctionnalité particulière sans pour autant "polluer" le flux principal.

Les branches servent à :

- travailler sur des fonctions spéciales
- Organiser le travail avec des environnements de développement distincts.



La commande **branch** permet de gérer tout ce qui a trait aux branches (ajout, liste, suppression, renommage).

Par défaut, il y a une branche sur le projet initial, qui est la branche *master*. C'est donc celle sur laquelle on travaille, de base.

Pour créer une nouvelle branche, tapez l'instruction :

```
$ git branch NomDeLaBranche
```

A noter : il faut choisir un nom en lien avec ce qui motive la création de la branche. Ici par exemple, on peut créer une branche « MenuAccueil » pour indiquer que l'on va travailler sur le menu de notre page d'accueil.



6

Branches

• Les branches

Pour lister les branches qui existent sur un projet, tapez l'instruction :

```
$ git branch
```

Résultat en console :

```
agnes@DESKTOP-KQAH1DD  
$ git branch  
MenuAccueil  
* master
```

On voit bien ici que l'on a 2 branches :

- MenuAccueil (que l'on vient d'ajouter)
- master (branche principale)

Le fait que la branche *master* apparaisse en couleur et avec un astérisque indique que l'on se situe sur cette branche.



6

Branches

• Les branches

Pour passer de la branche principale à une autre, on utilise la commande **checkout** :

```
$ git checkout MenuAccueil
```

A partir de là, les modifications qui seront faites s'affecteront à la branche en cours.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Mon site web</title>
</head>
<body>
  <h1>Bienvenue! </h1>
  <div id="page">
    <div class="logo"></div>
    <ul id="navigation">
      <li><a href=""></a></li>
      <li><a href=""></a></li>
      <li><a href=""></a></li>
      <li><a href=""></a></li>
    </ul>
  </div>
  <div>Article 1</div>
  <div>Article 2</div>
</body>
</html>
```

Ajout dans l'*index.html*

1
2

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (MenuAccueil)
$ git add index.html

agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (MenuAccueil)
$ git commit -m "mise en place du menu"
[MenuAccueil 1b334da] mise en place du menu
1 file changed, 10 insertions(+)

agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (MenuAccueil)
$ git log
commit 1b334da82827f738c14d301c0c9bbd78a3ec5423 (HEAD -> MenuAccueil)
Author: Agnès CEFii <a.rahard@cefii.fr>
Date:   Wed Mar 6 17:03:04 2019 +0100

    mise en place du menu

commit 5522cb3cfb7d9cb73c8f1f84f4c33537f5e38710 (master)
Author: Agnès CEFii <a.rahard@cefii.fr>
Date:   Wed Mar 6 16:37:40 2019 +0100

    modification du contenu de la page d'accueil

commit 93a923afe0a7c342ee958850383fd2142f8fdb9e
Author: Agnès CEFii <a.rahard@cefii.fr>
Date:   Wed Mar 6 16:25:53 2019 +0100

    initialisation du projet
```




6

Branches

- **Les branches**

Pour fusionner une branche avec une autre, ie reprendre les modifications faites de part et d'autre et les assembler dans une seule et même branche, on va se positionner dans la branche qui va recevoir les modifications (ici la branche *master*) puis utiliser la commande **merge**:

```
$ git merge NomDeLaBranche
```

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64
$ git merge MenuAccueil
Updating 5522cb3..1b334da
Fast-forward
 index.html | 10 ++++++++
 1 file changed, 10 insertions(+)
```

Ici il n'y a pas eu de difficulté particulière dans la mesure où il n'y a que des ajouts dans le script *index.html* (10 lignes insérées).

En revanche, il peut arriver qu'on ait des conflits entre différentes versions quand on veut fusionner 2 branches. Git signale alors qu'il y a un problème et il faudra traiter au cas pas cas les points litigieux.



6

Branches

- **Les branches**

En cas de conflit lors de la fusion des branches (exemple des modifs dans un même fichier dans chacune des branches), on aura un message d'erreur :

```
$ Auto-merging file.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

En parallèle de cela, on aura des indications sur ce qui pose problème dans le script concernés

```
<html>
<<<<<<< HEAD
  <head><title></title></head>
  <body>
    Erreur 404
  </body>
</html>

=====
  <head><title>Erreur 404</title></head>
  <body>
  </body>
</html>
>>>>>> dev1
```



6

Branches

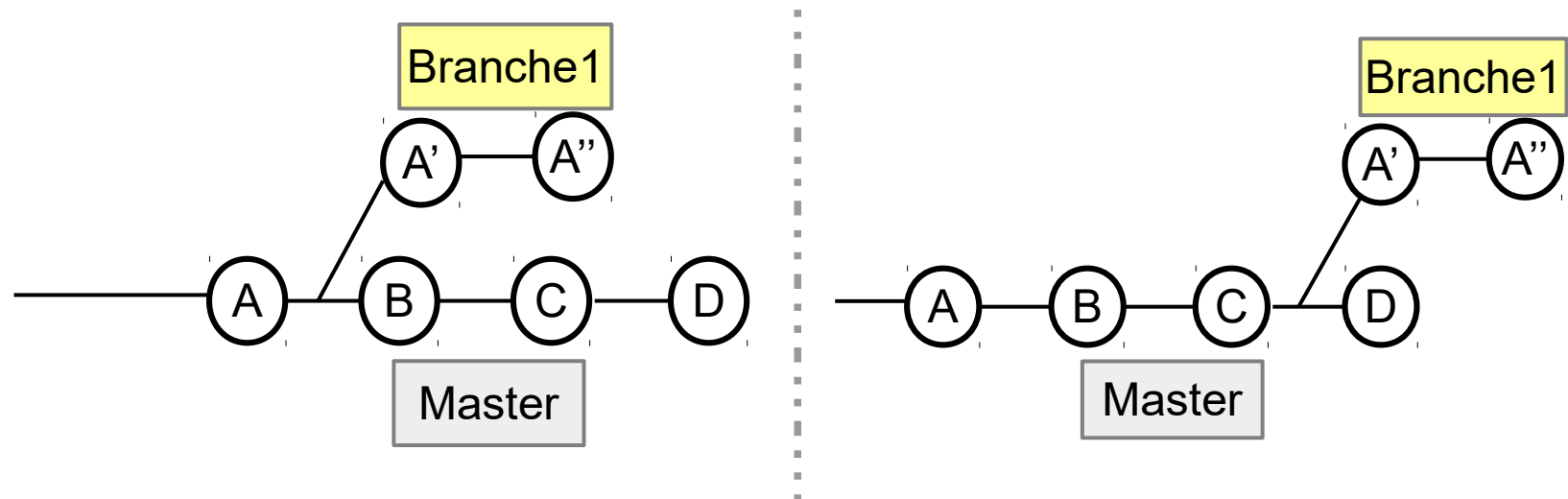
- **Les branches**

Il faut donc reprendre le code et faire le tri dans les différentes modifs afin de ne conserver que ce que l'on souhaite.

Ensuite on sauvegarde comme habituellement (add et commit).

Déplacer la base d'une branche

La commande **rebase** permet de déplacer une branche et de changer son commit de départ (soit la base sur laquelle elle a été conçue).





6

Branches

- **Les branches**

Pour supprimer une branche, tapez:

```
$ git branch -d NomDeLaBranche
```

Attention, pour supprimer une branche il ne faut pas qu'il y ait de commit en attente.

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/g  
$ git branch -d MenuAccueil  
Deleted branch MenuAccueil (was 1b334da).
```

Cette opération n'est pas obligatoire : on peut très bien laisser une branche en l'état et poursuivre le projet sur une autre.

Cependant, si l'on décide de ne plus travailler sur une branche, il est conseillé de la supprimer afin qu'il n'y ait pas d'ambiguïté sur l'utilité de cette dernière.

Les dépôts distants

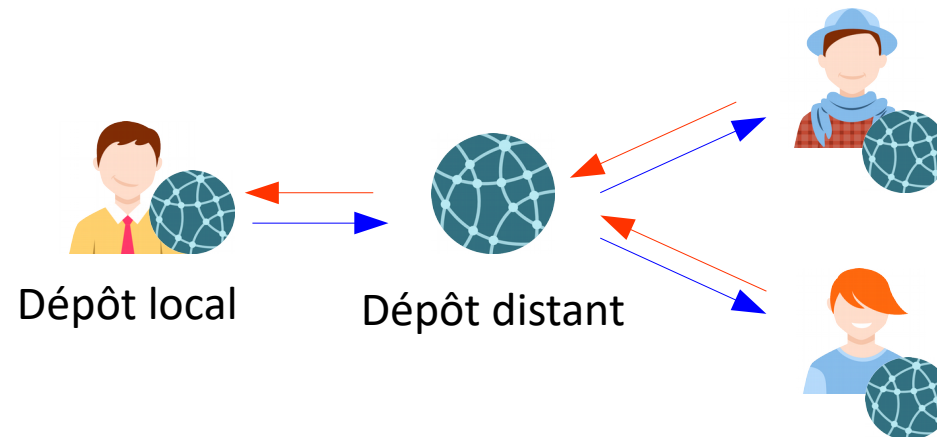


7

Dépôts distants

- **Les dépôts distants**

L'étape suivante est la **distribution du projet** sur un espace partagé afin que plusieurs personnes aient accès aux mêmes ressources et puissent ainsi collaborer sur ce projet.



Pour cela, le créateur du projet va déposer ses fichiers sur un **dépôt distant**.

Ensuite, les collaborateurs vont pouvoir télécharger ce projet à partir de ce dépôt.

Dans le cadre d'un travail collaboratif, chaque membre peut participer au projet soit en déposant des éléments, soit en les téléchargeant.



7

Dépôts distants

- **Les dépôts distants**

Pour la gestion de notre dépôt distant, nous allons utiliser **GitHub**, un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git.

Cette plateforme, très connue et utilisée dans le monde des développeurs, est open-source. Sa prise en main est relativement simple.

Dans un premier temps, il faut se créer un compte en allant sur cette page :



<https://github.com/join?source=header-home>

The screenshot shows the GitHub sign-up interface. At the top, there's a navigation bar with links like 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing'. A search bar and 'Sign in'/'Sign up' buttons are also present. The main heading is 'Built for developers'. Below it, a paragraph describes GitHub as a development platform. On the right, there's a sign-up form with fields for 'Username' (containing 'agnesCEFii'), 'Email' (containing 'you@example.com'), and 'Password' (masked with dots). A green 'Sign up for GitHub' button is at the bottom of the form. A small note at the bottom of the form states: 'By clicking "Sign up for GitHub", you agree to our terms of service and privacy statement. We'll occasionally send you account related emails.'



7

Dépôts distants

• Les dépôts distants

Une fois le compte créé, on accède à la plateforme. On va pouvoir créer un **repository** (un dépôt distant).

Sur cette page, il faut indiquer :

Create a new repository

A repository contains all project files, including the revision history.

The screenshot shows the GitHub 'Create a new repository' form. Three red circles with numbers 1, 2, and 3 are placed to the left of the form fields, indicating the required information:

- 1. Owner: agnesCEFii
- 2. Repository name: tutoGit
- 3. Description (optional): Tutoriel pour Git et GitHub

Below the description field, there are two radio button options for visibility:

- ☒ Public: Anyone can see this repository. You choose who can commit.
- ☐ Private: You choose who can see and commit to this repository.

At the bottom, there is a checkbox for 'Initialize this repository with a README' and two dropdown menus for 'Add .gitignore: None' and 'Add a license: None'.

1. le nom du repository
2. une description du projet
3. La visibilité : par défaut, le repository est *public*, tout le monde peut accéder au projet.
A noter que la visibilité *privée*, qui restreint l'accessibilité du projet aux seules personnes autorisées, est payante sur GitHub



7

Dépôts distants

- **Les dépôts distants**

Maintenant que le dépôt distant est créé sur GitHub, il faut créer le lien entre ce dépôt distant et notre dépôt local.

Pour cela, GitHub affiche les commandes à utiliser sur la page suivant la création d'un repository :

The screenshot shows the GitHub repository setup page. At the top, there's a navigation bar with links: Code, Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. Below this, a section titled "Quick setup — if you've done this kind of thing before" offers two options: "Set up in Desktop" and "HTTPS SSH". The "HTTPS SSH" option is selected, showing the URL `https://github.com/agnesCEFII/tutoGit.git`. Below this, a message says: "Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#)." The next section, "...or create a new repository on the command line", contains a code block with the following commands:

```
echo "# tutoGit" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/agnesCEFII/tutoGit.git
git push -u origin master
```

 The final section, "...or push an existing repository from the command line", contains a code block with the following commands:

```
git remote add origin https://github.com/agnesCEFII/tutoGit.git
git push -u origin master
```

 The code block for the second section is highlighted with an orange dashed border.

```
<> Code  Issues 0  Pull requests 0  Projects 0  Wiki  Insights  Settings
```

Quick setup — if you've done this kind of thing before

Set up in Desktop or **HTTPS SSH** `https://github.com/agnesCEFII/tutoGit.git`

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# tutoGit" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/agnesCEFII/tutoGit.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/agnesCEFII/tutoGit.git
git push -u origin master
```



7

Dépôts distants

- Les dépôts distants

On va donc taper ces instructions en ligne de commande sur notre terminal.

```
$ git remote add origin https://github.com/agnesCEFII/tutoGit.git
```

Pour vérifier que Git a bien intégré ce lien vers le dépôt distant, on peut utiliser la commande **git remote** qui va indiquer *origin* pour dire qu'il a bien trouvé le lien vers le dépôt distant.

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git remote add origin https://github.com/agnesCEFII/tutoGit.git

agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git remote
origin
```

Pour avoir la liste des dépôts distants et les chemins associés, tapez :

```
$ git remote -v
```

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git remote -v
origin https://github.com/agnesCEFII/tutoGit.git (fetch)
origin https://github.com/agnesCEFII/tutoGit.git (push)
```



7

Dépôts distants

- **Les dépôts distants**

Tout est donc en place pour dialoguer entre les dépôts local et distant sur notre projet.

L'envoi des fichiers du local vers le distant se fait grâce à la commande **git push** en indiquant le nom du dépôt et la branche à déposer.

Exemple :

```
$ git push -u origin master
```

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 1.01 KiB | 0 bytes/s, done.
Total 9 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/agnesCEFII/tutoGit.git
 * [new branch]      master -> master
```

L'envoi est terminé.



7

Dépôts distants

- **Les dépôts distants**

En allant sur la plateforme GitHub, on voit bien que le fichier index.html a été ajouté, avec tout l'historique du projet (ici 3 commits) :

The screenshot shows the GitHub interface for the repository 'agnesCEFii / tutoGit'. At the top, there's a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. Below the repository name, there are statistics: 1 Unwatch, 0 Stars, and 0 Forks. The main navigation bar includes Code, Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. The repository title is 'Tutoriel pour Git et GitHub'. Below this, there are statistics: 3 commits, 1 branch, 0 releases, and 1 contributor. A bar at the bottom shows the current branch as 'master' and a 'New pull request' button. The commit history shows a single commit by 'agnesCEFii' titled 'mise en place du menu' with the latest commit hash '1b334da' 5 hours ago. The file 'index.html' is listed with the same commit message and time. At the bottom, there's a prompt to 'Add a README'.



7

Dépôts distants

- **Les dépôts distants**

Ensuite, pour ajouter des collaborateurs, on va aller dans l'onglet **Settings** et ajouter les noms des collaborateurs que l'on veut associer à notre repository.

The screenshot shows the GitHub interface for the repository 'agnesCEFii / tutoGit'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below this is a navigation bar with tabs for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Insights', and 'Settings' (which is highlighted). On the left side, there is a sidebar with a list of settings: 'Options', 'Collaborators' (highlighted with an orange bar), 'Branches', 'Webhooks', 'Notifications', 'Integrations & services', 'Deploy keys', 'Moderation', and 'Interaction limits'. The main content area is titled 'Collaborators' and includes a link 'Push access to the repository'. It contains the text: 'This repository doesn't have any collaborators yet. Use the form below to add a collaborator.' Below this is a search bar with the placeholder text 'Search by username, full name or email address' and a note: 'You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.' There is an empty input field and an 'Add collaborator' button.



7

Dépôts distants

- **Les dépôts distants**

Enfin, pour faire l'opération inverse (du distant vers le local), on utilise la commande :

```
$ git pull origin master
```

Résultat en console :

```
agnes@DESKTOP-KQAH1DD MINGW64 /c/wamp64/www/git/projet (master)
$ git pull origin master
From https://github.com/agnesCEFII/tutoGit
* branch          master      -> FETCH_HEAD
Already up-to-date.
```

Ainsi pourra-t-on récupérer en local l'ensemble du projet modifié par d'autres collaborateurs qui auront déposé leur modif sur le dépôt distant.



7

Dépôts distants

- **Les dépôts distants**

En cas de décalage avec le dépôt distant, il est possible de forcer l'envoi avec **--force**

```
$ git push MonDepot --force
```

Utile en cas de problème (oubli d'une info sensible dans un commit, ...) cette option est à utiliser avec précaution car elle va modifier l'historique distant et affecter tous les collaborateurs.

Supprimer un dépôt distant

```
$ git remote rm MonDepot
```

Renommer un dépôt distant

```
$ git remote rename MonDepot MonDepot1
```



7

Dépôts distants



- **Les dépôts distants**

Fork

Un fork désigne une copie d'un dépôt.

Par défaut il n'est possible de faire de commit que sur un dépôt dont on est propriétaire.

La notion de fork permet d'avoir un dépôt sur lequel on a la permission d'écriture.

Pull request

Lorsqu'on travaille sur un fork, on peut proposer à l'auteur original du projet d'intégrer les modifications réalisées par le biais d'un ***pull request***.

Ce processus est géré de manière quasi automatique par GitHub qui propose un bouton « pull request ». Celui-ci envoie automatiquement un message au mainteneur pour l'informer de la requête qui lui est adressée sur ce projet.

Autre solution : lancer la commande ***git request-pull*** depuis un terminal et envoyer manuellement par e-mail le résultat au mainteneur de projet

Pour plus d'info sur ce thème, voir la [documentation officielle](#).

**Cloner un
projet**



8

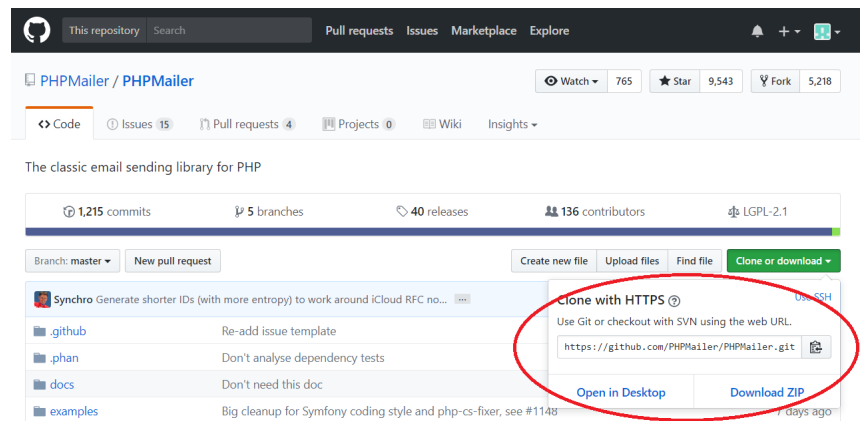
Cloner projet

- **Cloner un projet**

Cloner un dépôt de Git

Cloner un dépôt existant consiste à récupérer tout l'historique et tous les codes source d'un projet avec Git.

Se positionner sur la page GitHub du projet. On a la liste des fichiers et des derniers changements ainsi qu'un champ contenant l'adresse du dépôt.



Le clonage est une copie exacte : les deux dépôts sont les images parfaites l'une de l'autre, ils possèdent le même historique.

Suite au clonage, la branche master du clone fait référence à la branche master du cloné. Si plusieurs personnes travaillent sur un projet, elles peuvent ainsi en obtenir chacune une copie.



8

Cloner projet

• Cloner un projet

Dans le cas du PHPMailer, l'adresse du dépôt est :
<https://github.com/PHPMailer/PHPMailer.git>

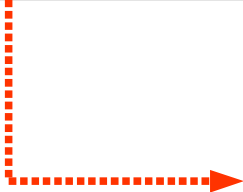
Il existe plusieurs protocoles :

- http:// et https://
- git://
- ssh://

L'utilisation du SSH permet de chiffrer les données pendant l'envoi et gère l'authentification des utilisateurs.

La commande pour cloner le projet PHPMailer en local est donc :

```
$ git clone https://github.com/PHPMailer/PHPMailer.git
```



```
Cloning into 'PHPMailer'...  
remote: Counting objects: 6081, done.  
remote: Compressing objects: 100% (20/20), done.  
remote: Total 6081 (delta 10), reused 14 (delta 3), pack-reused 6058  
Receiving objects: 100% (6081/6081), 4.22 MiB | 250.00 KiB/s, done.  
Resolving deltas: 100% (3978/3978), done.
```



8

Cloner projet

- **Cloner un projet**

A noter que sur GitHub, la plupart des dépôts sont en mode lecture seule (read only).

On peut télécharger les fichiers, effectuer des modifications en local mais pas les envoyer sur le serveur, afin d'éviter de détériorer les projets.

Pour apporter une modification sur un projet, il faut passer par l'un de ses développeurs pour qu'il télécharge et valide les modifications (ceci se fait par la création d'un Forks le plus souvent).

**Retour en
arrière**



9

Retour

• Retour en arrière

A partir du moment où on a sauvegardé les informations grâce au commit, on peut revenir en arrière pour revoir un ou plusieurs fichier(s) à une étape de sauvegarde.

Pour cela on dispose de la commande **checkout**

```
$ git checkout
```

La commande **checkout** permet de :

- Passer de branche en branche
- Revenir sur un fichier par rapport à un commit
- Revenir sur un commit

Pour revenir sur un fichier par rapport à un commit, il faut relever l'id du commit à restaurer et taper la commande :

```
$ git checkout 53717a0
```

On peut également cibler un fichier en particulier :

```
$ git checkout 53717a0 file.php
```



9

Retour

- **Retour en arrière**

Résultat : on retrouve le(s) fichier(s) dans l'état où ils étaient avant le commit indiqué.

En console il est indiqué le message suivant :

```
$ git checkout 53717a0
```

```
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:
```

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 53717a0... ajout des fichiers ignorés  
HEAD is now at 53717a0... ajout des fichiers ignorés
```



9

Retour

• Retour en arrière

Pour revenir à l'état de la branche principale, tapez :

```
$ git checkout master index.html
```

Attention le **checkout** permet de retrouver un état précédent pour observer les modifications faites sur un fichier mais on ne pourra pas repartir sur cet état antérieur.

Pour cela, il faut utiliser d'autres commandes.

Autre méthode avec la commande **revert** qui permet d'inverser un commit en particulier (sans affecter les modifications apportées par les suivants).

Cette commande va donc :

- défaire ce qui avait été fait au moment du commit
- créer un nouveau commit.

Le **revert** n'altère pas l'historique mais va ajouter un nouveau commit d'inversion (les lignes ajoutées seront supprimées, les fichiers supprimés seront recréés...).

```
$ git revert f400b45
```

A noter : le *revert* étant un commit, on peut faire un *revert* d'un *revert*...



9

Retour

- **Retour en arrière**

Le **reset** supprime tout ou partie des fichiers de la zone de staging, sans supprimer les modifications.

```
$ git reset
```

Là aussi on peut cibler un fichier en particulier :

```
$ git reset file.php
```

Le **reset** permet de revenir en arrière jusqu'à un commit particulier.

```
$ git reset f400b45
```

Il réinitialise la zone de staging (espace tampon avant le commit) tout en laissant le dossier de travail en l'état.

L'historique est perdu, c'est à dire que les *commits* suivant sont perdus, mais pas les modifications.



9

Retour

• Retour en arrière

Le *reset* permet surtout de nettoyer l'historique en resoumettant un commit unique à la place de commit trop éparses.

Utilisé avec le paramètre *hard*, le *reset* renvoie le dossier de travail au niveau du dernier commit.

Toutes les modifications non commitées seront perdues (à utiliser avec précaution).

```
$ git reset --hard
```

Si l'on part du HEAD, on peut remonter en arrière en utilisant le caractère ^ autant de fois que de niveau à remonter :

```
$ git reset HEAD^^
```

Astuces



10

Astuces

• Astuces

Sauvegarde des modifications avec vi

```
$ git commit
```

Ceci a pour effet d'ouvrir « vi », éditeur de texte bien connu comme étant un modèle d'ergonomie spécialement adaptée aux développeurs...

Voici comment procéder pour saisir un texte :

- taper « i » pour passer en mode insertion
- Saisir le texte
- Sortir en sauvegardant en tapant Echap puis « :wq »

Résultat en console:

```
$ git commit  
[master (root-commit) 8b52198] premier commit  
1 file changed, 1 insertion(+)  
create mode 100644 readme.md
```



10

Astuces

• Astuces

Exclusion de fichier

On peut indiquer à Git de ne pas prendre en compte certains fichiers en les notant dans un script **.gitignore**

```
log/*  
*.tmp
```

Exemple : ici on ne sauvegarde pas les fichiers du dossier « log » ni ceux avec l'extension « tmp ».

Afficher les logs

On peut limiter le nombre de réponse (ici à 2 logs) :

```
$ git log -n 2
```



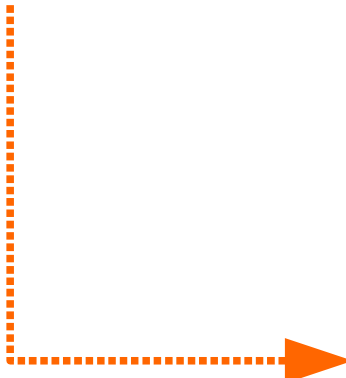
10

Astuces

- **Astuces**

voir l'historique des commits sur un fichier en particulier :

```
$ git log --oneline -p file.php
```



```
$ git log --oneline -p 404.html
5e57ef8 (HEAD -> master) Modif HTML
diff --git a/404.html b/404.html
index e69de29..f2ed5f5 100644
--- a/404.html
+++ b/404.html
@@ -0,0 +1,10 @@
+<html>
+  <head>
+    <title></title>
+  </head>
+  <body>
+  </body>
+</html>
+
\ No newline at end of file
53717a0 ajout des fichiers ignorés
diff --git a/404.html b/404.html
new file mode 100644
index 0000000..e69de29
```



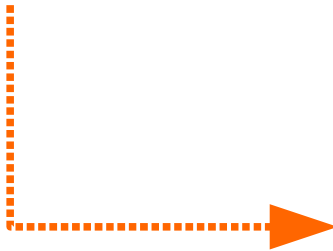
10

Astuces

- **Astuces**

Lister les différences entre 2 commits :

```
$ git diff
```



```
diff --git a/readme.md b/readme.md
index 7c4d713..3168ca0 100644
--- a/readme.md
+++ b/readme.md
@@ -1,2 +1,3 @@
-Lisez moi
-nouvelle modif
\ No newline at end of file
+nouvelle modif
+sdqsqs
+d
```



10

Astuces

- **Astuces**

Ajouter des modifications à un commit

Si on veut ajouter une petite modification à un commit déjà fait, on utilise l'argument `--amend`.

```
$ git commit --amend
```

Cela permet d'ajouter les fichiers en staging dans le commit précédent, ce qui est pratique en cas d'oubli et évite de faire plusieurs commits sur une même partie du développement.

Envoi de toutes les branches sur le serveur distant

```
$ git push MonDepot --all
```


Interfaces graphiques



11

GitKraken

- **Interfaces Graphiques**

L'environnement natif de Git est le **terminal**.

Les nouvelles fonctionnalités y apparaissent en premier et c'est seulement à la ligne de commande que l'on dispose de tout le pouvoir de Git.

Mais le texte pur n'est pas toujours le meilleur choix pour toutes les tâches.

Une représentation visuelle est préférable et certains utilisateurs sont beaucoup plus à l'aise avec une interface graphique.

Source Tree



Développé par Atlassian, **Source Tree**, est un des logiciels les plus poussés qui vous permettra de quasiment tout gérer sans passer par le terminal.

ungit



C'est un client web développé avec nodejs (qu'il faudra installer en local), disponible sur **GitHub**.



11

Logiciels Git



• Interfaces Graphiques

GitHub Desktop

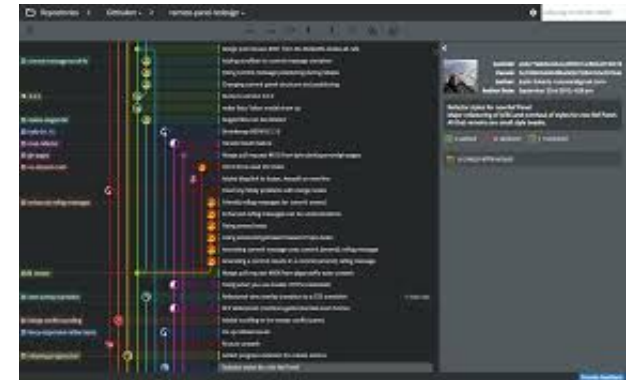
GitHub Desktop ne propose que les fonctions de bases :

- Création de commit et de revert
- Gestion des branches
- Gestion du remote, Push & Pull

GitKraken

Développé par Axosoft Gitkraken, **GitKraken** dispose de nombreuses fonctionnalités:

- Gestion des branches
- Intégration de Git Flow
- Gestion des remotes
- Gestion des tags
- Connexion à Github
- Rebase, Cherry Pick, Ammend



Webographie



9

Webographie

- **Webographie**

<https://git-scm.com/book/fr/v1/>
<https://gitexplorer.com/>

CEFii



www.cefii.fr



git

GitHub



Fin du tutoriel