TP git

Sommaire

Thierry Vaira - <tvaira@free.fr> - version v0.1 - 24/08/2021 - tvaira.free.fr

Objectif : Utiliser git (en ligne de commande et avec GitHub) en tant que développeur seul.

1. Présentation

Git est un logiciel de **gestion de versions décentralisé** (DVCS). C'est un logiciel libre créé par **Linus Torvalds** en 2005. Il s'agit maintenant du logiciel de gestion de versions le plus populaire devant Subversion (`svn`) qu'il a remplacé avantageusement.

Site officiel : https://git-scm.com/ + Ressources

La fonction principale de Git est de suivre les différentes versions d'un projet. Un projet est un ensemble de fichiers (généralement en texte).

Un *commit* (ou instantané) représente un ensemble cohérent de modifications sur le projet. Le *commit* est l'élément central de Git.

2. Travail demandé

Utiliser le Guide Git BTS SN.

Rédiger le compte-rendu dans Google Classroom en indiquant à chaque fois les commandes exécutées et le résultat pertinent obtenu (sinon remplacer par des ⋯ pour les résultats de commande non significatifs) complété éventuellement d'un commentaire.

Utiliser obligatoirement le module **Code Blocks** dans Google Docs pour formater les commandes du *shell* et le code source C++.

2.1. Séquence n°0 : Installation/Configuration

Cette séquence a pour objectif d'installer et configurer Git en ligne de commande (CLI).

2.1.1. Installation

Sous GNU/Linux Ubuntu :

```
$ sudo apt-get install git

$ git --version
git version X.Y.Z
```

1 . Identifier la version de Git.

2.1.2. Configuration

Configuration du compte :

```
$ git config --global user.name "<votre nom>"
$ git config --global user.email "<votre email>"
```

Choix de l'éditeur de texte :

```
$ git config --global core.editor vim
```

Activation de la coloration :

```
$ git config --global color.diff auto
$ git config --global color.status auto
$ git config --global color.branch auto
```

Visualiser la configuration

```
$ cat $HOME/.gitconfig

$ git config --list
```

2 . Fournir la configuration de Git.

2.2. Séquence n°1a : les commandes de base

Cette séquence a pour objectif de découvrir les commandes de base nécessaires pour réaliser la majorité des activités avec Git en local :

- configurer et initialiser un dépôt,
- commencer et arrêter le suivi de version de fichiers,
- indexer et valider des modifications.

On abordera aussi :

- le paramétrage de Git pour ignorer certains fichiers,
- revenir sur les erreurs rapidement et facilement,
- parcourir l'historique du projet et voir les modifications entre deux validations.

2.2.1. Initialiser un dépôt local

Création d'un répertoire :

```
$ mkdir tp-git-sequence-1
mkdir: création du répertoire 'tp-git-sequence-1'

$ cd ./tp-git-sequence-1
```

Initialisation d'un dépôt git :

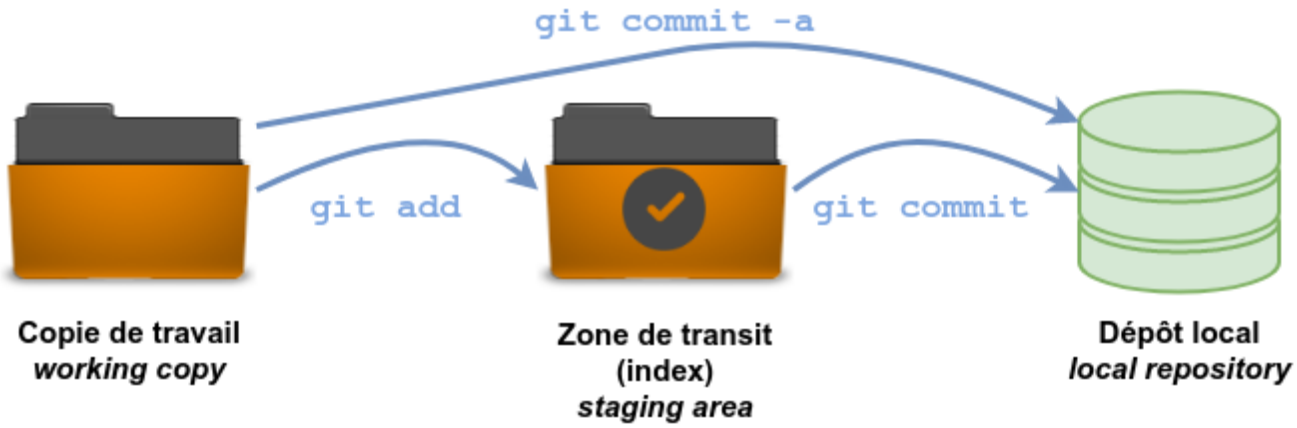
```
$ git init
Dépôt Git vide initialisé dans $HOME/tp-git-sequence-1/.git/
```

3 . Lister le contenu du répertoire contenant le dépôt Git.

2.2.2. Travailler avec git

L'utilisation standard de Git se passe comme suit :

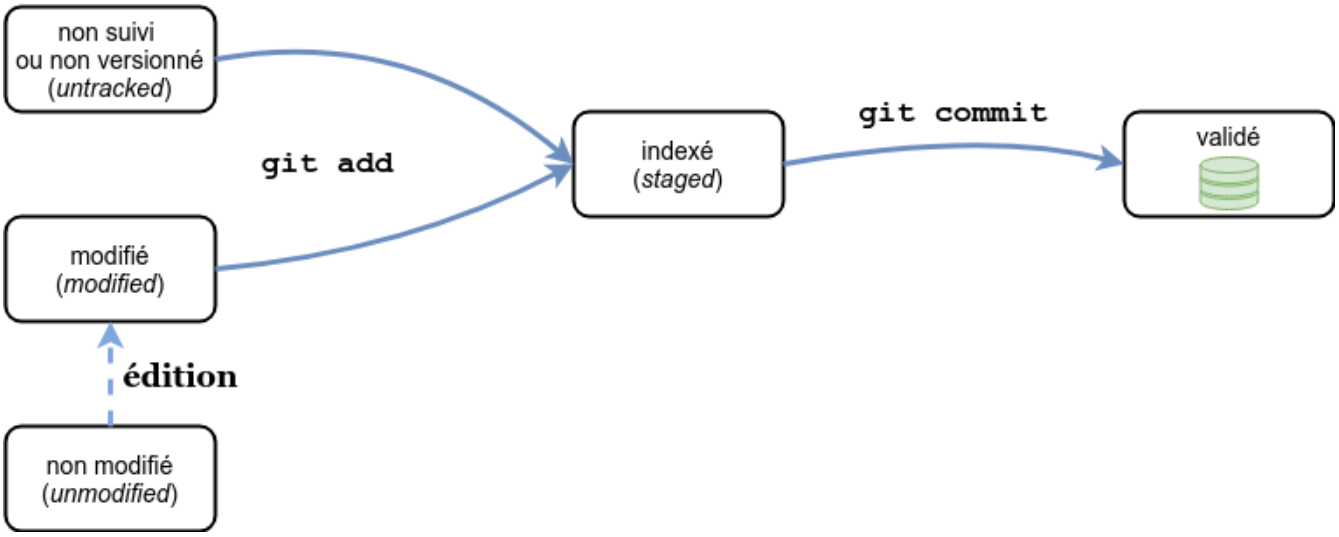
- on édite des fichiers dans le répertoire de travail (*working directory*) ;
- on indexe les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (*staging area*) ;
- on valide les modifications, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans le dépôt local (*local repository*).



Lien : https://ndpsoftware.com/git-cheatsheet.html

Les différents états d'un fichier :

- non suivi ou non versionné (*untracked*) : aucun instantané existe pour ce fichier
- non modifié (*unmodified*) : non modifié depuis le dernier instantané
- modifié (*modified*) : modifié depuis le dernier instantané mais n'a pas été indexé
- indexé (*staged*) : modifié et ajouté dans la zone d'index
- validé : une version particulière d'un fichier



Pour obtenir l'état des fichiers du répertoire de travail (*working directory*), on utilise (très souvent)

la commande `git status` :

La commande git status :

```
$ git status --help
$ git help status

# Exemples :
$ git status
$ git status -s
$ git status -b
$ git status --long
$ git status -v
```

4 . Indiquer les fichiers actuellement versionnés dans le dépôt Git.

5 . Quel est le nom de la branche principale actuelle du dépôt Git ?

6 . Ajouter un nouveau fichier vide `bienvenue.cpp` au dépôt Git.

7 . Modifier le fichier `bienvenue.cpp` avec le contenu suivant :

```
// TODO Indiquer ce que fait le programme

int main()
{
   // TODO Afficher un message de bienvenue

   return 0;
}
```

8 . Vérifier la fabrication de l'exécutable `bienvenue`. Tester.

9 . Valider les mofications sur le dépôt.

10 . Créer un fichier `.gitignore` qui permet d'ignorer les fichiers issus de la fabrication (exécutable, fichiers objets, ...).

Liens :

- gitignore
- Une collection de modèles .gitignore

11 . Ajouter le fichier `.gitignore` au dépôt Git.

12 . Modifier le programme principal `bienvenue.cpp` avec le contenu ci-dessous. Vérifier la fabrication de l'exécutable `bienvenue`. Tester.

```
// Affiche un message de bienvenue

#include <iostream>

int main()
{
   std::cout << "Bienvenue le monde !" << std::endl;

   return 0;
}
```

13 . Montrer les différences obtenues sur le fichier `bienvenue.cpp`.

14 . Valider les changements.

15 . Créer un fichier `README` avec le contenu ci-dessous. Ajouter le au dépôt.

```
# Bienvenue

Programme C++ qui affiche "Bienvenue"
```

16 . Modifier maintenant le fichier `README` avec le contenu ci-dessous et valider les changements.

```
# Bienvenue

Programme C++ qui affiche "Bienvenue le monde !"
```

17 . Renommer le fichier `README` en `README.md`. Valider les changements.

18 . Montrer l'historique des *commits* en affichant chaque *commit* sur une seule ligne.

19 . En utilisant un rebasage interactif, regrouper les *commits* (concernant le fichier en Markdown) en un seul avec le message : "Ajout du fichier README.md". Montrer l'historique des *commits* en affichant chaque *commit* sur une seule ligne.

20 . Étiqueter la version `1.0`. Créer une archive `tp-git-sequence-1-nom.zip` à partir de cette version (en précisant votre `nom` comme identifiant). La fournir avec le compte-rendu du TP.

2.3. Séquence n°1b : les branches

Cette séquence a pour objectif de découvrir les branches et de travailler avec.

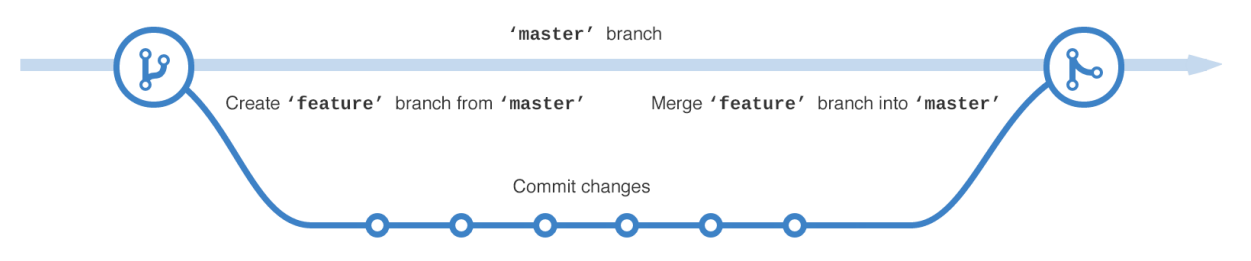
- créer une nouvelle branche,
- basculer sur une branche existante,
- fusionner une branche,
- supprimer une branche.

Définition

Une branche représente une ligne de développement indépendante : créer une branche signifie diverger de la ligne principale de développement et continuer à travailler sans impacter cette ligne.

On crée des nouvelles branches depuis `master` ou `main` à chaque nouvelle fonctionnalité ou nouvelle modification qu'il faut apporter au projet. Git permet de gérer plusieurs branches en parallèle et ainsi de cloisonner les travaux et d'éviter ainsi de mélanger des modifications du code source qui n'ont rien à voir entre elles.

En gardant une branche `master` ou `main` saine, on conserve ainsi une version du logiciel prête à être livrée à tout instant puisqu'on ne fusionne dedans que lorsque le développement d'une branche est bien terminé.



Techniquement, une branche dans Git est simplement un pointeur déplaçable vers un *commit*.

Liens :

- Manuel de référence en français dans le chapitre "Les branches avec Git"
- Wikilivre Git - Branches

2.3.1. Travailler avec les branches

Il est important de noter que lorsque l'on change de branche avec Git, les fichiers du répertoire de travail sont modifiés. Si la copie de travail ou la zone d'index contiennent des modifications non validées qui sont en conflit avec la branche à extraire, Git n'autorisera pas le changement de branche. Le mieux est donc d'avoir une copie de travail propre au moment de changer de branche.

1 . Créer une branche `fonction-bienvenue`.

2 . Lister les branches existantes.

3 . Basculer sur la branche `fonction-bienvenue`.

4 . Travailler dans la branche `fonction-bienvenue`, en créant les fichiers ci-dessous.

fonction-bienvenue.h

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

void afficherBienvenue();

#endif // FONCTION_BIENVENUE_H
```

fonction-bienvenue.cpp

```
#include "fonction-bienvenue.h"
#include <iostream>

void afficherBienvenue()
{
   std::cout << "Bienvenue le monde !" << std::endl;
}
```

bienvenue.cpp

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
   afficherBienvenue();

   return 0;
}
```

Makefile

```
TARGET := bienvenue
MODULE := fonction-bienvenue

CXX = g++ -c
LD = g++ -o
RM = rm -f
CXXFLAGS = -Wall -std=c++11
LDFLAGS =

$(info Fabrication du programme : $(TARGET))

all : $(TARGET)

$(TARGET): $(TARGET).o $(MODULE).o
    $(LD) $@ $(LDFLAGS) $^

$(TARGET).o: $(TARGET).cpp $(MODULE).h
    $(CXX) $(CXXFLAGS) $<

$(MODULE).o: $(MODULE).cpp $(MODULE).h
    $(CXX) $(CXXFLAGS) $<

.PHONY: clean

clean:
    $(RM) *.o

cleanall:
    $(RM) *.o $(TARGET)

rebuild: clean all
```

5 . Tester le "travail" et valider les modifications.

6 . Montrer l'historique des *commits* en affichant chaque *commit* sur une seule ligne.

7 . Basculer sur la branche `master` (ou `main`).

8 . Fusionner la branche `fonction-bienvenue` dans `master` (ou `main`). Quelle est le type de fusion réalisée par Git ?

9 . Supprimer la branche `fonction-bienvenue`. Lister les branches existantes.

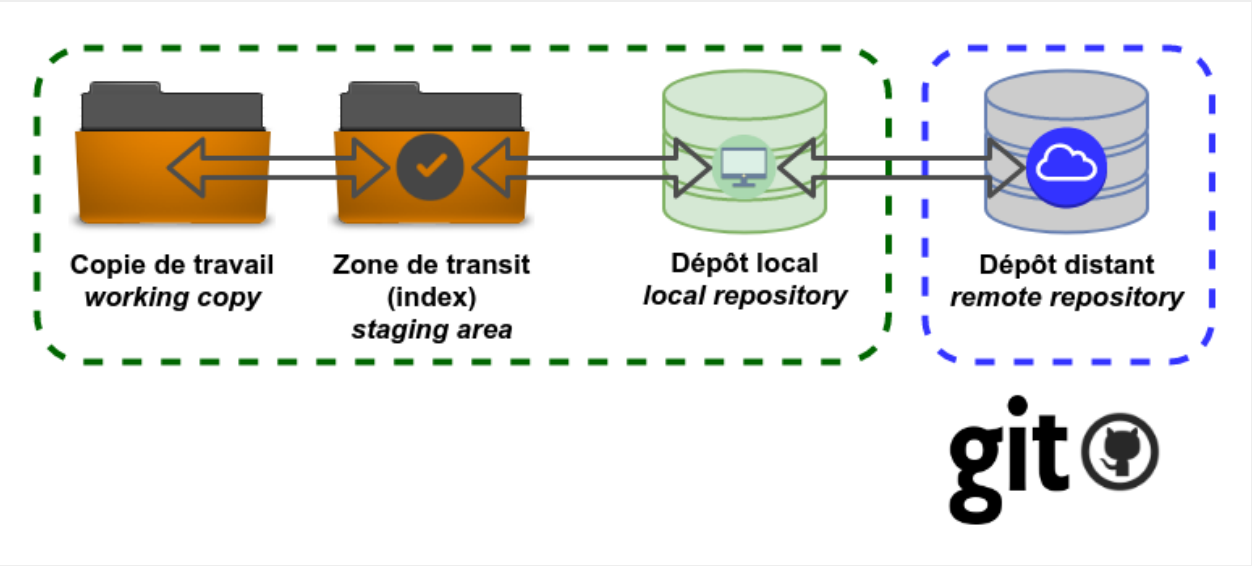
10 . Montrer l'historique des *commits* en affichant chaque *commit* sur une seule ligne. Lister les fichiers du répertoire de travail.

2.4. Séquence n°1c : le dépôt distant

Cette séquence a pour objectif de mettre en oeuvre l'utilisation d'un dépôt distant (*remote repository*) sur GitHub.

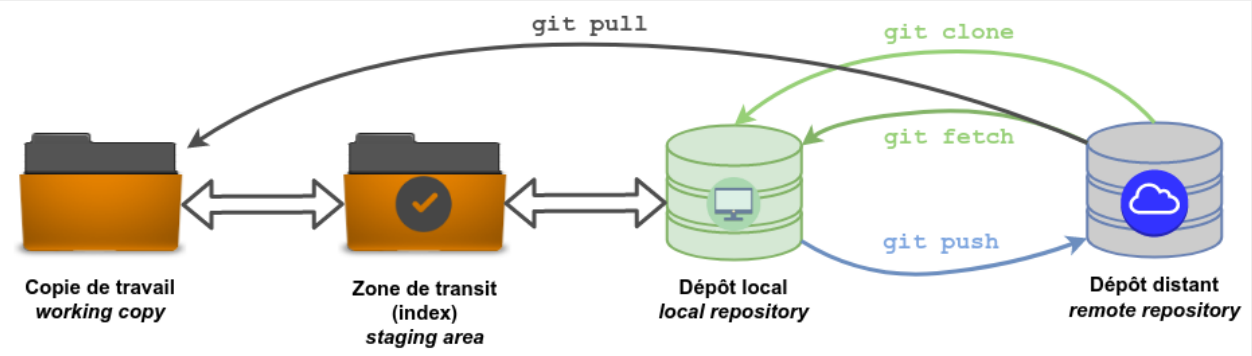
Définition

Un dépôt distant est un dépôt hébergé sur un serveur, généralement sur Internet. Cela permet de sauvegarder, récupérer et surtout partager un projet.



Liste : https://git.wiki.kernel.org/index.php/GitHosting

L'utilisation d'un dépôt distant introduit des commandes spécifiques comme `git clone`, `git push`, `git fetch` et `git pull`.



2.4.1. GitHub

GitHub est un service web d'hébergement (lancé en 2008) et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git.

Site officiel : https://github.com/

1 . Créer un compte GitHub.

2 . Créer des clés SSH ou un jeton personnel HTTPS.

Liens :

- https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
- https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token

2.4.2. Travailler avec un dépôt distant

3 . Créer un nouveau dépôt du même nom que votre dépôt local.

4 . Associer le dépôt distant et le dépôt local.

```
$ git remote add origin https://github.com/<utilisateur>/<depot>.git
```

5 . Renommer la branche `master` en `main` si besoin.

6 . Synchroniser les deux emplacements.

7 . Lister les dépôts distants.

8 . Modifie le fichier `README.md` sur le dépôt local et valider les changements.

```
# Bienvenue

Programme C++ qui affiche "Bienvenue le monde !" en utilisant la fonction
`afficherBienvenue()`.
```

9 . Publier les modifications sur le dépôt distant. Vérifier sur l'interface web de GitHub.

10 . Éditer le fichier `README.md` (pour ajouter le code source) directement à partir l'interface web de GitHub.

Exemple pour insérer du code source dans un fichier au format Markdown :

````
L'exécution d'une commande Linux :

```sh
$ make rebuild
Fabrication du programme : bienvenue
rm -f *.o
g++ -c -Wall -std=c++11 bienvenue.cpp
g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
g++ -o bienvenue  bienvenue.o fonction-bienvenue.o

$ ./bienvenue
Bienvenue le monde !
```

Le contenu d'un fichier source C++ :

```cpp
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

void afficherBienvenue();

#endif // FONCTION_BIENVENUE_H
```
````

Lien : Syntaxe d’écriture et de formatage de base sur GitHub

11 . Mettre à jour le dépôt local. Montrer le contenu du fichier `README.md` à partir de la ligne de commande.

cat est une commande Unix standard permettant de concaténer des fichiers ainsi que d'afficher leur contenu sur la sortie standard.

bat est un clone de la commande cat qui possède la coloration syntaxique.

```
▶ bat test.md
       File test.md
   1   # Markdown example
   2
   3   Note how we can correctly syntax-highlight the
   4   code blocks *inside* the Markdown document.
   5
   6   Python example:
   7
   8   ``` python
   9   cmd = "bat"
  10   print("Hello from {}".format(cmd))
  11   ```
  12
  13   Ruby example:
  14
  15   ``` ruby
  16   cmd = "bat"
  17   puts "Hello from #{cmd}"
  18   ```
```

12 . Créer localement le *tag* `1.1` et publier le sur le dépôt distant.

13 . Créer une version livrable "*release*" sur GitHub.

14 . Créer un nouveau dépôt distant `tp-cplusplus` sur GitHub.

15 . Cloner le dépôt distant `tp-cplusplus`.

N'effectuez pas le clonage dans le répertoire de travail actuel !

16 . Lister le contenu du dépôt local obtenu.

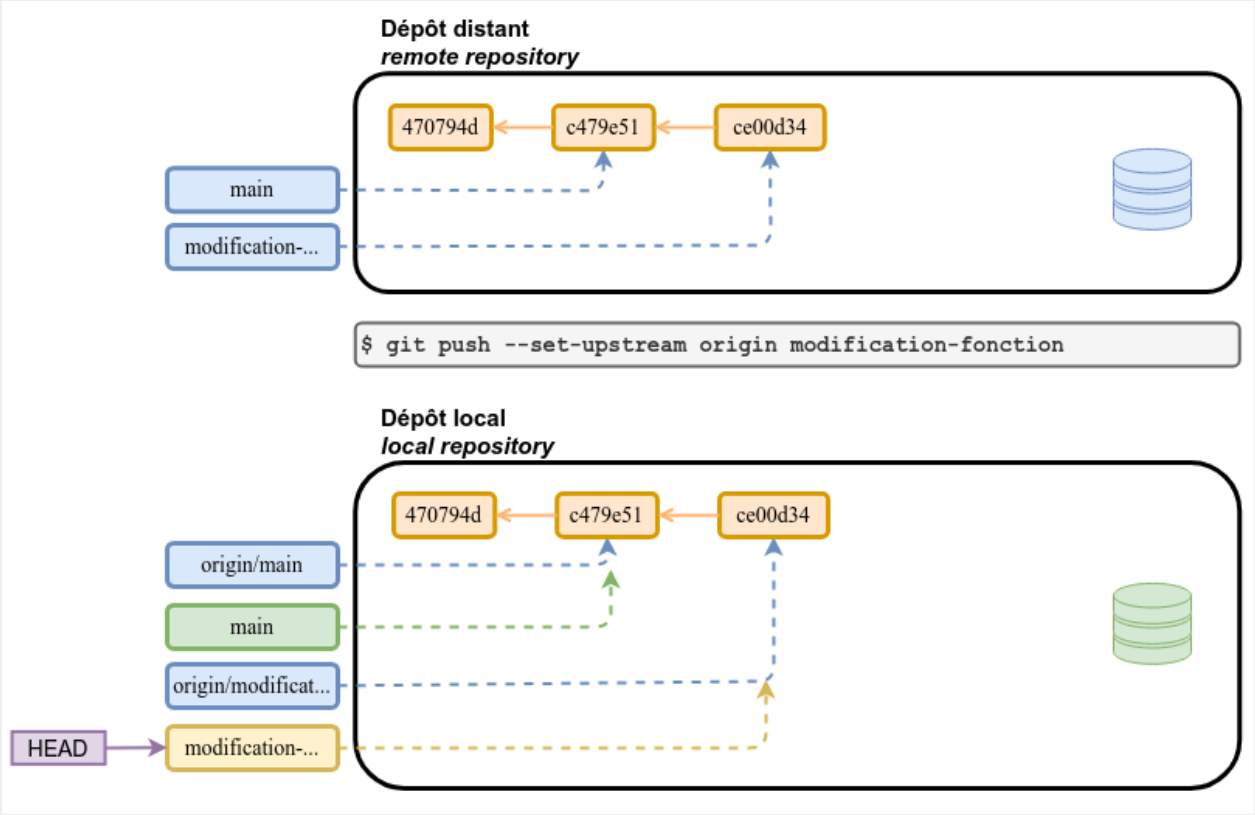
Vous pouvez conserver ce dépôt distant pour les Travaux Pratiques C++ ou le supprimer.

2.5. Séquence n°1d : les branches de suivi

Cette séquence a pour objectif de découvrir l'utilisation des branches de suivi.

Définition

Une branche de suivi (*tracking branch*) est une branche locale qui est en relation directe avec une branche distante (*upstream branch*).



Les branches de suivi peuvent servir :

- à sauvegarder le travail sur la branche dans un dépôt distant
- partager le travail sur la branche avec d'autres développeurs

Utiliser le dépôt `tp-git-sequence-1` pour cette séquence.

2.5.1. Travailler avec une branche de suivi

1 . Créer une branche `modification-fonction` et on basculer dessus.

2 . Travailler sur le code en modifiant les fichiers ci-dessous.

fonction-bienvenue.h

```
#ifndef FONCTION_BIENVENUE_H
#define FONCTION_BIENVENUE_H

#include <string>

void afficherBienvenue(std::string message="Bienvenue le monde !");

#endif // FONCTION_BIENVENUE_H
```

fonction-bienvenue.cpp

```
#include "fonction-bienvenue.h"
#include <iostream>

void afficherBienvenue(std::string message/*="Bienvenue le monde !"*/)
{
   std::cout << message << std::endl;
}
```

bienvenue.cpp

```
// Affiche un message de bienvenue

#include "fonction-bienvenue.h"

int main()
{
   afficherBienvenue();

   return 0;
}
```

3 . Tester le "travail" et valider les modifications.

4 . Créer maintenant une branche de suivi pour cette branche.

5 . Lister les branches (avec l'option `-vv`).

6 . Fusionner localement la branche dans la branche principale. Quelle est le type de fusion réalisée par Git ?

6 . Lister les branches qui ont été fusionnées.

8 . Publier (pousser) les modifications de la branche principale sur le dépôt distant.

9 . Afficher l'historique des *commits* sous forme de graphe.

10 . Supprimer la branche locale et distante. Pourquoi peut-on supprimer cette branche ?

11 . Créer une branche thématique directement dans GitHub pour effectuer un correctif sur le fichier `README.md`.

12 . Récupèrer la branche distante et basculer dessus. Est-elle automatiquement une branche de suivi ?

13 . Modifier le fichier `README.md` pour documenter le changement de code par exemple.

14 . Valider les changements et les publier sur le dépôt distant.

15 . Créer une *Pull Request* dans GitHub pour fusionner les derniers changements dans la branche principale.

Définition

Pull Request peut être traduit par « Proposition de révision » (PR) : c'est-à-dire une demande de modification ou de contribution.

Les *Pull Requests* sont une fonctionnalité permettant de proposer une contribution vers un dépôt central.

Les *Pull Requests* facilitent aussi la collaboration des développeurs sur un projet : informer les membres de l'équipe qu'un « travail » (une fonctionnalité, une version livrable, un correctif, ...) est terminé et nécessite une révison de code. Pendant cette révision de code, les développeurs peuvent discuter de la fonctionnalité (commenter le code, poser des questions, ...), proposer des adaptations de la fonctionnalité en publiant des *commits* de suivi et au final fusionner le « travail » quand il est approuvé.

16 . Finaliser la *Pull Request* dans GitHub.

17 . Basculer la branche principale et mettre à jour le dépôt local. Montrer le contenu du fichier `README.md` à partir de la ligne de commande.

18 . Afficher l'historique des *commits* sous forme de graphe.

19 . Supprimer complètement la branche de suivi.

20 . Créer localement le *tag* `1.2` et publier le sur le dépôt distant.

2.6. Séquence n°1e : l'EDI Visual Studio Code

L'objectif de cette séquence est d'utiliser Git avec l'environnement de développement intégré Visual Studio Code.

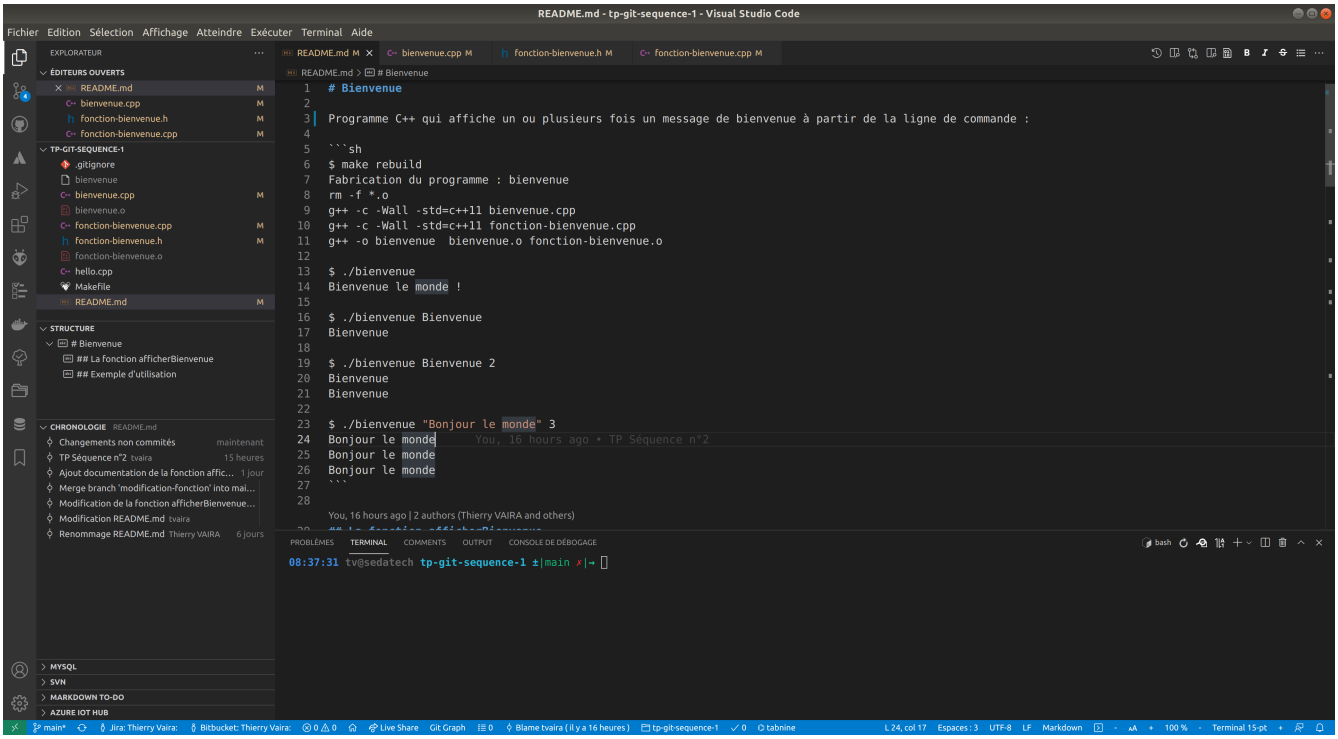
2.6.1. Visual Studio Code

Visual Studio Code est un éditeur de code extensible développé par Microsoft© pour Windows©, Linux et macOS©.

C’est un éditeur de code multi-plateforme, open source et gratuit, supportant une dizaine de langages (C/C++, Java, PHP, Python, Javascript, ...).

1 . Installer (si besoin) et utiliser Visual Studio Code (éventuellement avec certaines extensions mentionnées dans le guide) pour cette séquence :

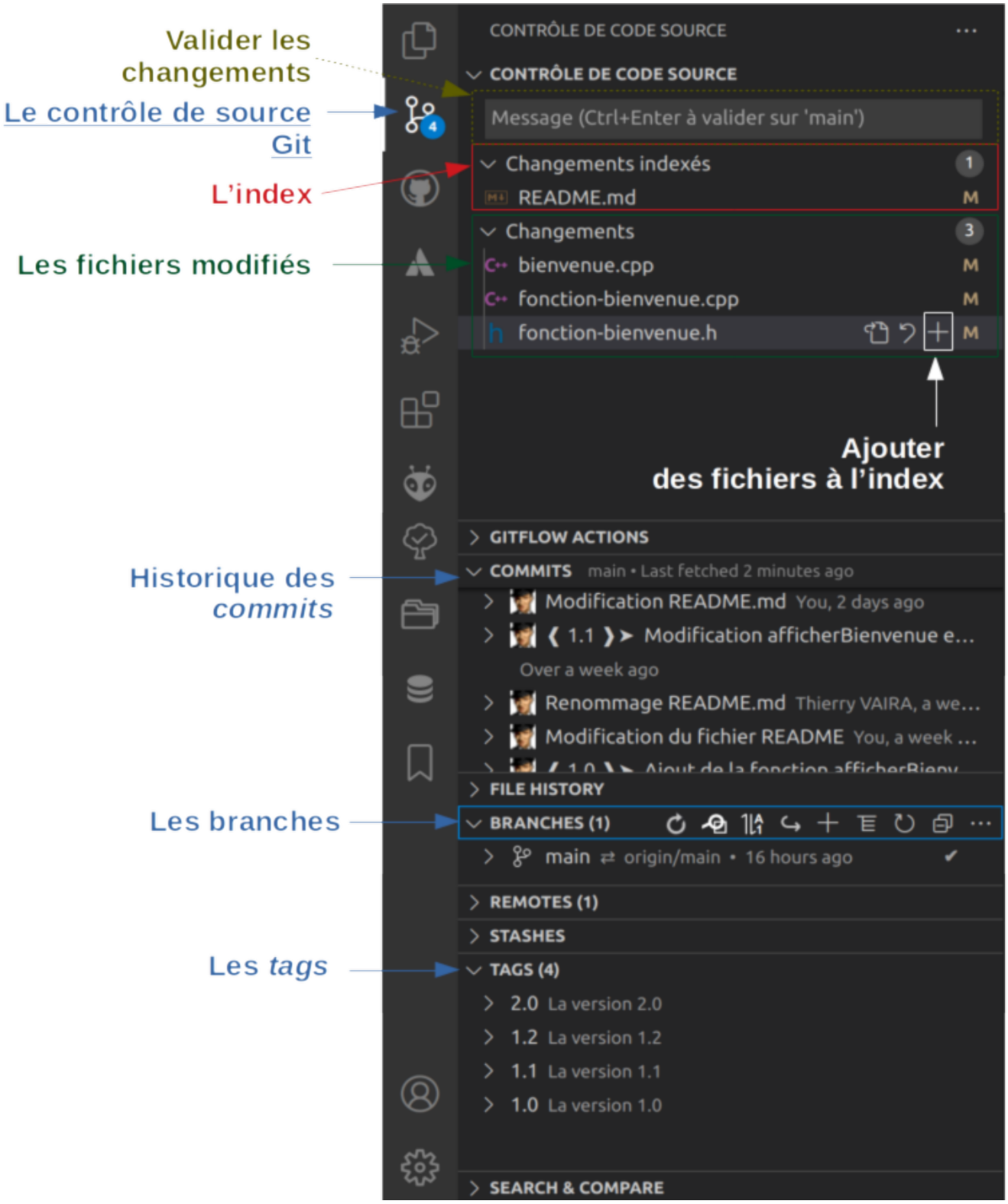
- Download
- Setup
- Getting Started



Lien : http://tvaira.free.fr/projets/activites/activite-visualcode.html

2.6.2. Travailler avec un EDI

Le contrôle de source (SCM) dans Visual Studio Code :



Liens :

- https://code.visualstudio.com/docs/editor/versioncontrol
- https://code.visualstudio.com/docs/editor/github

Ici, vous pouvez envisager de provoquer volontairement un conflit de fusion et de le résoudre !

2 . Modifier la fonction `afficherBienvenue()` pour qu'elle recoive un argument supplémentaire

nommé `nbAffichage` (de type `int`) qui représente le nombre de fois que le `message` doit s'afficher (`1` fois par défaut). Modifier le fichier `README.md` en conséquence.

3 . Créer un *tag* 1.3 pour cette version. Fournir une archive `.zip` de ce *tag*.

4 . Vous devez modifier le programme principal (et le fichier `README.md`) pour qu'il recoive éventuellement en arguments de la ligne de commande, dans cet ordre :

- le message de bienvenue à afficher
- le nombre de fois que le message de bienvenue doit s'afficher

```
$ make rebuild
Fabrication du programme : bienvenue
rm -f *.o
g++ -c -Wall -std=c++11 bienvenue.cpp
g++ -c -Wall -std=c++11 fonction-bienvenue.cpp
g++ -o bienvenue  bienvenue.o fonction-bienvenue.o

$ ./bienvenue
Bienvenue le monde !

$ ./bienvenue Bienvenue
Bienvenue

$ ./bienvenue Bienvenue 2
Bienvenue
Bienvenue

$ ./bienvenue "Bonjour le monde" 3
Bonjour le monde
Bonjour le monde
Bonjour le monde
```

Lire : le document annexe-fonction-main.pdf pour l'utilisation des arguments de la ligne de commande et Saisie avec cin pour des exemples de conversions C++.

5 . Créer un *tag* 2.0 pour cette version. Fournir une archive `.tar.gz` de ce *tag*.

6 . Fournir la démarche Git utilisée pour cette séquence.

3. Ressources Git

- Guide Git BTS SN
- Manuel de référence
- Livre Pro Git en français
- Livre Git Community Book en français
- Wikilivre Git en français

- Wikipedia Git
- Git Handbook sur Github

4. Les commandes principales

Git est un ensemble de commandes indépendantes dont les principales sont :

- `git init` crée un nouveau dépôt ;
- `git clone` clone un dépôt distant ;
- `git add` ajoute le contenu du répertoire de travail dans la zone d'index pour le prochain *commit* ;
- `git status` montre les différents états des fichiers du répertoire de travail et de l'index ;
- `git diff` montre les différences ;
- `git commit` enregistre dans la base de données (le dépôt) un nouvel instantané avec le contenu des fichiers qui ont été indexés puis fait pointer la branche courante dessus ;
- `git branch` liste les branches ou crée une nouvelle branche ;
- `git checkout` permet de basculer de branche et d'en extraire le contenu dans le répertoire de travail ;
- `git merge` fusionne une branche dans une autre ;
- `git log` affiche la liste des *commits* effectués sur une branche ;
- `git fetch` récupère toutes les informations du dépôt distant et les stocke dans le dépôt local ;
- `git push` publie les nouvelles révisions sur le dépôt distant ;
- `git pull` récupère les dernières modifications distantes du projet et les fusionne dans la branche courante ;
- `git tag` liste ou crée des *tags* ;
- `git stash` stocke de côté un état non commité afin d'effectuer d'autres tâches.

Liens :

- AIDE MÉMOIRE GITHUB GIT PDF
- Git CHEATSHEET
- Git Cheat Sheet

Obtenir de l'aide :

```
$ git help
$ git --help
$ man git

$ git help <commande>
$ git <commande> --help
$ man git-<commande>
```

Un thème pour `bash` qui intègre Git : oh-my-bash

Thierry Vaira - <tvaira@free.fr> - version v0.1 - 24/08/2021 - tvaira.free.fr