TD 1 — Introduction à l'utilisation de Git

Git est un outil de gestion de versions, très utilisé en développement informatique. Il permet de suivre l'historique des modifications, de revenir à des versions antérieure (de manière temporaire ou définitive), de travailler sur plusieurs fonctionnalités en parallèle en évitant les conflits de modifications, et de travailler à plusieurs développeurs·euses sur un même projet informatique.

Un des objectifs de cette ressource R2.03 Qualité de développement est de vous faire une introduction à l'utilisation de Git. Il s'agit là d'une *première découverte* : le logiciel Git est extrêmement complet et il est possible d'en avoir un usage très avancé. Nous nous contentons ici d'aborder les éléments fondamentaux.

Ce TD est élaboré comme un *tutoriel* : lisez le document dans l'ordre et exécutez les instructions pas à pas.

Git a été initialement développé par Linus Torvalds, qui est aussi à l'origine du noyau système Linux. À l'heure actuelle, Git est de très loin le gestionnaire de versions le plus utilisé dans les projets informaiques. Savoir s'en servir est indispensable.

1 Installation et configuration

Git est disponible sur la plupart des systèmes d'exploitation. Sur les systèmes UNIX, il s'installe avec le gestionnaire de paquets habituel. Pour l'installer sur Windows, rendez-vous sur la page https://gitforwindows.org/ et cliquez sur Downloads. Depuis le dépôt Github, téléchargez le fichier `Git-*-64-bit.exe` dans sa version la plus récente. Puis suivez les instructions d'installation.

Différence entre Git et Github

Git est le logiciel qui gère les versions. Github est une plateforme en ligne sur laquelle on peut rendre disponible des dépôts Git pour travailler à plusieurs sur un même projet. Vous reviendons plus précisément sur cette différence un peu plus tard.

1.1 Modes d'utilisation de Git

Git peut s'utiliser de deux manière différentes.

En mode local Les modifications ont lieu sur votre machine, et l'historique des versions concerne uniquement le projet présent sur votre ordinateur.

En mode distant Vous travaillez en équipe sur un projet informatique, qui est disponible sur un *dépôt distant* (par exemple sur la plateforme Github, mais ce n'est pas la seule). Vous téléchargez le dépôt sur votre machine locale, puis vous développez vos fonctionnalités localement. Lorsque votre développement est terminé, vous pouvez envoyer vos modifications sur le dépôt commun. Les autres développeurs·euses pourront télécharger vos modifications et les synchroniser avec leur code local.

Dans cette introduction, nous allons principalement travailler *en mode local*. Vous devez seulement savoir que Git fournit, en mode distant, de nombreux outils pour gérer et résoudre les éventuels

conflits de modifications.

1.2 Configuration

Lorsque l'installation est terminée, exécutez Git. Cela ouvre un terminal : ce comportement est normal car *Git s'utilise en ligne de commande*. La première étape consiste à configurer Git.
Lancez les commandes suivantes :

Terminal
```
user@host:~$ git config --global user.name "votreNomDUtilisateur"
user@host:~$ git config --global user.email "prenom.nom@etu.u-pec.fr"
```

Ces deux opérations sont nécessaires pour que Git puisse fonctionner sur votre machine, mais elles ne sont pas très utiles en mode local. Elles servent surtout pour travailler sur un dépôt distant, car elles permettent de savoir qui a apporté quelles modifications dans le code. Pour votre nom d'utilisateur, vous pouvez mettre apremière lettre de votre prénom, et votre nom de famille (exemple : `c_pages`).
Vérifiez votre configuration avec la commande :

Terminal
```
user@host:~$ git config --list
user.name = c_pages
user.email = clement.pages@u-pec.fr
```

1.3 Création d'un dépôt Git

Dans votre répertoire de travail, créez les dossiers `R203_QDdev‣TD1_IntroGit` et placez-vous dans ce répertoire :

Terminal
```
user@host:~$ cd Documents/
user@host:~$/Documents/ mkdir R203_QDev/
user@host:~$~/Documents/R203_QDev/
user@host:~$ mkdir TD_IntroGit/
user@host:~$~/Documents/R203_QDev/TD_IntroGit/
```

Sur EPREL, on vous fournit un fichier `index.html`, que l'on utilisera comme exemple pour nos manipulations avec Git. Téléchargez ce fichier et placez-le dans votre répertoire `R203_QDdev‣TD1_IntroGit ‣`.

Nous allons maintenant *créer un dépôt Git* dans notre répertoire de travail :

Terminal
```
user@host:~$~/Documents/R203_QDev/TD_IntroGit/ git init
astuce: […]
Dépôt Git vide initialisé dans ~/R203_QDev/TD/TD_IntroGit/.git/
```

Pour le moment, nous pouvons ignorer les lignes `astuce`. La dernière ligne indique qu'un dépôt a été créé. Cela signifie qu'un dossier, nommé `.git‣` a été créé dans le répertoire de travail. C'est ici que Git va enregistrer l'historique de tout le projet.

Nom de dossier commençant par un point

Sous les systèmes UNIX, les dossiers et fichiers dont le nom commence par un point sont *cachés* : ils n'apparaissent pas lorsque l'on utilise la commande `ls`. Ainsi, le dépôt Git est simplement un dossier caché placé à la racine du projet.

2 Commits, historique, staging

2.1 Votre premier commit

Nous alllons travailler avec un projet fictif pour illustrer l'utilisation de Git. L'objectif n'est pas de développer, mais de *simuler* une situation où l'on travaille sur un projet, sur lequel on modifie un code existant.

Lancez la commande `git status`. Vous obtenez la sortie suivante :

Terminal

```
Sur la branche master

Aucun commit

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        index.html

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents
  (utilisez "git add" pour les suivre)
```

Analysons cette sortie ligne à ligne.

- La première ligne indique la *branche* de travail actuel. Nous y reviendrons plus tard.
- La mention `Aucun commit` indique que vous n'avez pas ajouté les fichiers modifiés au dépôt.
- On vous donne ensuite la liste des fichiers non suivis.

Lorsque vous développez, vous allez modifier vos codes sources. Lorsque les modifications vous semblent convaincantes (vous avez terminé de développer une fonctionnalité, vous souhaitez valider un test, etc.), vous pouvez *enregistrer une copie* de votre travail, pour revenir dessus plus tard. Cette opération s'appelle un *commit*. Pour le moment, notre dépôt ne contient aucun commit.

Nous souhaitons ajouter le fichier `index.html` aux fichiers suivis :

Terminal

```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git add index.html
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
        nouveau fichier : index.html
```

On observe que le fichier `index.html` est maintenant pris en compte. Lorsque nous allons faire le prochain commit, les modifications seront sauvegardées dans le dépôt :

Terminal

```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git commit -m "Premier commit :
inclusion du fichier index.html"
[master (commit racine) a8fb55c] Premier commit : inclusion du fichier index.html
 1 file changed, 34 insertions(+)
 create mode 100644 index.html
```

Un commit est donc une *sauvegarde* de votre travail à un instant donné. Cela permet d'en garder une copie. Plus tard, vous pourrez revenir à ce commit pour le consulter, ou pour rétablir votre code (par exemple, suite à des essais infructueux). L'attribut `-m` permet d'indiquer une courte description du commit, pour expliquer les modifications apportées dans le code.

Suivi de fichiers avec Git

Lorsque l'on est en train de développer, les fichiers peuvent avoir plusieurs statuts Git, par rapport au commit précédent.

Fichiers non modifiés Rien ne se passe pour les fichiers qui n'ont pas été modifiés depuis le précédent commit.

Fichiers non indéxés Vous avez modifié un fichier source, par exemple `contact.html`, mais vous n'avez pas appliqué la commande `git add contact.html`. Le fichier est actuellement *non suivi*, ce qui veut dire que les modifications apportées ne seront pas copiées dans le dépôt lors du prochain commit.

Fichiers indexés Lorsque l'on utilise la commande `git add contact.html`, le fichier est placé dans ce que l'on appelle le *staging*[a] Lors du prochain commit, c'est la version dans le staging qui sera enregistrée.

a. Que l'on peut approximativement traduire par « zone de transit ».

2.2 Historique des modifications

Chaque fois que l'on fait un nouveau commit, il est enregistré dans le dépôt (i.e. dans le dossier `.git`) et il est possible de voir l'historique des modifications :

Terminal

```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git log
commit a8fb55c6c1afdb14d579acf1ed719529515d1a18 (HEAD -> master)
Author: c_pages <clement.pages@u-pec.fr>
Date:   Mon Jan 26 08:46:24 2026 +0100

    Premier commit : inclusion du fichier index.html
```

Exercice 1. **1)** Modifiez le fichier `index.html` à votre convenance (l'objectif n'est pas de réviser le HTML mais d'utiliser les commandes Git, des modifications mineures suffisent).

2) Exécutez la commande `git status`. Quelles informations obtenez-vous ?

3) Nous souhaitons maintenant sauvegarder les modifications apportées. Quelles commandes faut-il utiliser ?

On suppose que vous avez maintenant une copie de travail à jour, avec deux commits : le premier commit, et celui que vous avez effectué lors de l'exercice 1.

Apportez d'autre modifications à votre fichier `index.html`, en ajoutant un code HTML invalide. Par exemple, on pourra introduire le paragraphe suivant à la fin de la page :

index.html

```
<p>Lorem ipsum dolor sit amet Morbi id interdum neque. Suspendisse suscipit
sodales orci, a imperdiet dolor ultrices quis. Nam sed leo et metus condimentum
eleifend nec vitae nunc. Vivamus blandit fringilla velit id vehicula. Suspendisse
eleifend eros commodo neque mollis maximus. Interdum et malesuada fames ac ante
ipsum primis in faucibus. Suspendisse <emph>potenti.</p></emph>
```

L'état actuel du dépôt est le suivant :

Terminal

```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications
   dans le répertoire de travail)
        modifié :         index.html

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou
"git commit -a")
```

Comme le code HTML n'est pas correct, on souhaite simplement l'effacer et revenir à la version enregistrée lors du dernier commit :

Terminal

```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git restore index.html
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

2.3 Ajout d'un nouveau fichier

On ajoute maintenant un fichier CSS à notre projet. Créez un fichier `design.css` et ajoutez-y les directives CSS de votre choix[1]. On pensera également à modifier le ficher `index.html` pour y intégrer le fichier style.

La commande `git status` indique maintenant que deux fichiers sont non-suivis : `index.html` et `design.css`. Ajoutez-les avec la commande : `git add index.html design.css`.

Ajouter tous les fichiers

Lorsque l'on souhaite ajouter *tous* les fichiers en attente au prochain commit, on peut utiliser la commande `git add --all`. Elle est utile lorsque l'on a modifié beaucoup de fichiers.

1. Là encore, l'objectif n'est pas de s'entrainer en CSS : quelques directives mineures suffisent.

3 Les branches

3.1 L'art de faire plusieurs choses à la fois

Les commits permettent de manipuler l'historique et de garder traces des modifications passées. Git est muni d'une autre fonctionnalité essentielle : les branches.

Lorsque l'on développe une nouvelle fonctionnalité, on souhaite garder une image du projet dans son état actuel, et développer les nouveautés de manière parallèle. Ensuite, on pourra fusionner les deux branches. Cela permet de ne pas introduire de bugs et de garder une version *stable* du projet, en découpant les différentes taches de développement.

Par défaut, la branche principale s'appelle `master` :

Terminal
```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch
* master
```

On suppose que l'on souhaite ajouter une nouvelle page à notre site, `apropos.html`. Pour développer cette page, on crée une nouvelle banche :

Terminal
```
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch page_apropos
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch
* maser
  page_apropos
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git switch page_apropos
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch
  master
* page_apropos
```

À partir de maintenant, toutes les modifications, les commits, etc. *ne se font que sur la branche* `page_apropos`. La branche `master` va rester dans sont état actuel.

Exercice 2. **1)** Créez la page `apropos.html` et ajoutez-y du contenu. Faites régulièrement des commits pour suivre votre travail.

2) Exécutez la commande `git log` pour avoir l'historique des modifications effectuées sur la branche courante (i.e. `page_apropos`).

3) Avec la commande `git status`, vérifiez que votre copie de travail est propre, et que tous les fichiers qui doivent l'être sont correctement commités.

Vous travaillez actuellement sur la branche `page_apropos`.

4) Revenez sur la branche `master` avec la commande `git switch master`.

5) Exécutez la commande `git log`. Comparez avec l'historique obtenu à la question 2). Que remarquez-vous ?

Les modifications apportées sur la branche `page_apropos` n'apparaissent pas dans la branche `master`. Lorsque la création de la page `apropos.html` est terminée, on peut fusionner la branche `page_apropos` dans la branche `master` :

```
Terminal
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git merge page_apropos
Mise à jour c84ecf3..8ea442f
Fast-forward
 apropos.html | 4 ++++
 1 file changed, 4 insertions(+)
```

La branche `page_apropos` n'est plus utile maintenant, on peut la supprimer :

```
Terminal
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch -d page_apropos
```

3.2 Revenir dans le passé

Supposons que l'on souhaite revenir à une version antérieure du code, soit parce qu'elle était plus fonctionnelle, soit pour la consulter. Commençons par afficher l'historique du dépôt :

```
Terminal
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git log
commit 8ea442fb14d7011c93c0451878db43c3b1709358 (HEAD -> master, dev)
Author: cpages <contact@clementpages.fr>
Date:   Mon Jan 26 11:48:21 2026 +0100
    Ajout du fichier design.css

commit c84ecf3a121e8f69eaba4bb56a52cb81d51d906d
Author: cpages <contact@clementpages.fr>
Date:   Mon Jan 26 11:38:04 2026 +0100

    Ajout de la page apropos.hhtml

commit e8ca76d096fdb45b8ce69445c984010c3f516631
Author: cpages <contact@clementpages.fr>
Date:   Mon Jan 26 09:03:53 2026 +0100

    Ajout d'une section au fichier index.html

commit a8fb55c6c1afdb14d579acf1ed719529515d1a18
Author: cpages <contact@clementpages.fr>
Date:   Mon Jan 26 08:46:24 2026 +0100

    Premier commit : inclusion du fichier index.html
```

Chaque commit est repéré par son identifiant (la longue suite de lettres et de chiffres) unique. On peut revenir à un état antérieur du projet en utilisant la commande suivante :

```
Terminal
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git checkout e8ca76d
Note : basculement sur 'e8ca76d'.
user@host:~/Documents/R203_QDev/TD_IntroGit/$ git branch
* (HEAD détachée sur e8ca76d)
master
```

Bien utiliser les identifiants des commits

Les identifiants utilisés ici sont ceux des commits effectués à titre d'exemple. Vous devez évidemment adapter les commandes aux identifiants des commits sur *votre* dépôt Git !

Notez par ailleurs qu'il n'est pas nécessaire de recopier l'identifiant du commit en entier : les premiers caractères suffisent.

La commande `git checkout <id_commit>` crée une branche temporaire, qui permet de *consulter* le code tel qu'il était au moment du commit choisi. Si l'on souhaite retravailler ce code, on doit d'abord créer une branche spécifique avec la commande `git switch -c <nom_nouvelle_branche>`. Ce dispositif permet de revenir dans le passé et de modifier du code antérieur *sans perdre le travail effectué entre temps* ! On pourra ensuite, si on le souhaite, fusionner les branches pour garder les modifications que l'on souhaite.

4 Quelques fioritures utiles

4.1 Les alias de commandes

Premier élément d'ergonomie : le terminal fourni avec Git est le terminal Bash d'UNIX. Vous pouvez donc utiliser toutes les commandes que vous connaissez (`ls`, `cd`, `cp`, `grep`, etc.). On peut remonter l'historique des commandes avec les flèches ↑ et ↓ du clavier, et compléter automatiquement une commande avec la touche Tab.

Lorsque l'on utilise Git, on fait très souvent appel aux mêmes commandes (log, commit, branch, etc.). Il est possible de créer des alias pour les commandes plus souvent utilisées. On recommande de créer (au moins) les alias suivants :

Terminal
```
user@host:~/$ git config --global alias.co checkout
user@host:~/$ git config --global alias.br branch
user@host:~/$ git config --global alias.ci commit
user@host:~/$ git config --global alias.st status
user@host:~/$ git config --global alias.sw switch
```

Le paramètre `--global` indique que l'alias est applicable à tous les utilisateurs et toutes les utilisatrices de la machine [2]. Avec ces alias, par exemple la commande `git st` est équivalente à `git status`, ce qui est plus rapide à écrire.

On peut également utiliser l'alias `git lg` défini ci-dessous, qui permet d'afficher les logs des 10 derniers commits sur une seule ligne (cela prend moins de place sur l'écran, et évite de remonter trop loin dans le passé) :

Terminal
```
user@host:~/$ git config --global alias.lg "log --oneline --max-count=10"
```

2. Ce qui n'aucune importance ici, puisque vous êtes seul·e sur votre ordinateur.

Définissez vos propres alias !

Vous pouvez, si vous le souhaitez, créer vos propres alias pour les commandes que vous utilisez le plus souvent.

4.2 Intégration de Git et outils graphiques

Si vous programmez avec VSCode[3], vous pouvez intégrer le terminal Bash (et donc Git) depuis le bouton de pilotage des terminaux ouverts.

Enfin, Git est fourni avec une interface graphique rudimentaire, qui peut aider à visualiser l'état actuel du dépôt. On l'ouvre avec la commande `gitk`.

4.3 Le fichier `.gitignore`

Pendant le développement, il est fréquent de manipuler des fichiers ou des dossiers *annexes*, qui ne font pas partie du code (par exemple, les fichiers temporaires, certains fichiers de configuration, des logs, des fichiers écrits par les programmes, etc.).

Ces fichiers n'ont donc pas besoin d'être suivis par Git et ne feront jamais partie des commits. Pour indiquer à Git que certains fichiers n'ont pas besoin d'être suivis, on crée, à la racine du dépôt[4], un fichier nommé `.gitignore`. Ce fichier va contenir l'ensemble des fichiers et dossiers à ignorer. Sa syntaxe est très simple : un élément par ligne.

.gitignore

```
logs/* # Le dossier logs/ et tout son contenu
*.csv  # Tous les fichiers csv
test.txt # Le fichier test.txt
```

On peut donner des directives très avancées au fichier `.gitignore`, cf. la documentation pour en savoir davantage.

Le mot de la fin

Nous utilisons ici Git en mode *local*, sur notre machine. Le plus souvent, Git s'utilise en se connectant à un dépôt *distant*, permettant de travailler à plusieurs sur le même projet, tout en ayant une bonne gestion des versions et des conflits de modifications.

Le bon usage des branches

Sur la branche principale (i.e. la branche `master`), on ne garde que *le code correctement testé et dans une version stable*. On crée une branche `dev`, sur laquelle on va développer les nouvelles fonctionnalités. Chaque fonctionnalité est développée dans une branche spécifique, que l'on fusionne ensuite dans `dev`. Lorsque la branche `dev` est considérée comme suffisamment stable et aboutie, on la fusionne dans `master`.

3. Vous de devriez pas vous infliger tant de souffrances !
4. C'est-à-dire au même endroit que là où se trouve le dossier `.git`.

Ainsi, la branche principale `master` contient, à tout instant, un code en bon état de fonctionnement, même si toutes les fonctionnalités ne sont pas présentes.

Git en mode distant

L'utilisation en mode distant nécessite d'être *parfaitement à l'aise* avec les commandes vues dans ce TD, car la gestion distante fait appel à des notions plus avancées. L'idée est très simple, mais les cas d'usages peuvent être assez nombreux.

- On travaille sur un projet disponible dans un dépôt distant.
- On effectue une copie *locale* du dépôt, et on travaille sur le code source *sur sa machine locale*, pour apporter les fonctionnalités demandées. Localement, on peut créer des commits, des branches, etc., de manière usuelles.
- Lorsque le travail est terminé, on *pousse* le nouveau code développé sur le dépôt distant.

La problématique est de réussir à gérer des éventuelles modifications qui auraient eu lieu *en simuultané* plusieurs développeurs·ses. Lorsque l'on pousse des modifications sur un dépôt, Git vérifie si d'autres ajouts on eu lieu et vous avertit le cas échéant. Il faudra alors gérer les éventuels conflits, avant de pousser votre code à votre tour.

Pour limiter au maximum les conflits, la bonne pratique consiste à *créer une branche par fonctionnalité*, et à pousser cette branche sur le dépôt. Ainsi, le code commun n'est pas impacté. La ou le chef·fe de projet fusionne les branches lorsqu'elles ont été validées, pour maintenir une branche principale « saine ».

À retenir

- Git est l'outil indispensable à savoir utiliser pour gérer les versions dans un projet informatique.
- Il s'utilise en ligne de commande.
- Il faut avoir compris comment suivre un fichier et l'ajouter dans le staging, comment restaurer un fichier.
- Vous devez savoir naviguer dans l'historique du projet avec `git checkout`.
- Vous devez savoir créer, fusionner, supprimer des branches et gérer les conflits éventuels.