GIT

http://git-scm.com/book/fr/v2

INSTALLATION DE GIT

1. Se créer un compte depuis le site **GitHub.com**
2. Télécharger l'exécutable :

Linux -> http://git-scm.com/download/linux, suivre les instructions selon sa distrib Linux, Debian :

```
# apt-get install git-all
```

Windows -> 2 possibilités : checker le lien de téléchargement sur http://git-scm.com/download/win (renvoie vers https://gitforwindows.org en ce moment) ou télécharger la version « desktop » https://desktop.github.com

3. Dans la console : ou dans les fichiers **/etc/gitconfig** ou **~/.gitconfig** (si besoin de les modifier)

```
git config --global user.name USERNAME
git config --global user.email EMAIL
git config --global color.diff auto
git config --global color.status auto
git config --global colo.branch auto
# Authentifier un serveur git distant sous WSL
git config --global credential.helper « /mnt/c/Program \
Files/Git/mingw64/libexec/git-core/git-credential-managerexe »
git config --global core.autocrlf false # turn off the autoline
endings feature as this can hide vertain issues especially when
working with both Linux & Windows
```

Relancer la console, taper **git**. Si tout s'est bien déroulé, la notice d'utilisation s'affiche.

CRÉER UN REPO

2 possibilités :

```
git clone URL # Si repo déjà existant et fourni
```

ou créer le repo sur GH, puis :

```
git init
git remote add origin git@github.com:USERNAME/REPO.git
git remote add origin https://github.com/USERNAME/REPO.git   } ou
git add .
git commit –am « Message de commit »
git push –u origin master # Au moins la 1ere fois, ensuite on peut
se passer de –u origin master
```

FAIRE LE 1ER COMMIT

Créer un nouveau **repository** (répertoire de travail dans lequel on fait des dépôts) ou utiliser un dossier existant et se positionner dedans avec la console. Activer le dossier comme repo (sinon Git ne peut pas savoir que c'est un repo)

```
git init
```

Pour gérer un repo, Git génère un **index** de tous les fichiers à suivre du repo. À chq nouveau fichier du repo, il faut l'ajouter à l'index.

```
git add FILE
```

Pour gagner du temps, on peut ajouter tout le répertoire courant :

```
git add .
```

Lorsqu'on a bossé sur ses files et que le repo est modifié, on demande d'enregistrer les modifications

```
git commit –m « message descriptif de la modif apportée sur le repo »
```

Par la suite, on pourra saisir

```
git commit –am « message descriptif de la modif apportée sur le repo »
```

L'option –a demande de mettre à jour les fichiers déjà existants dans son index (donc, il faut déjà avoir une 1ere fois « git add » son FILE)

On vérifie le statut du repo (s'il y a des fichiers à indexer, commiter, pusher…)

```
git status
```

On peut faire tout ceci depuis le site Github, voir https://guides.github.com/activities/hello-world/ pour tuto 1er commit.

LIRE L'HISTORIQUE

Pour remonter l'historique et comprendre ce qui a été fait, on affiche la liste de tous les commits réalisés :

```
git log
```

Le sens de lecture du **log** est le suivant : du commit le plus récent en haut de liste au commit le plus ancien en bas de liste. Chaque commit est répertorié avec son **SHA** (identifiant unique), son **auteur**, sa **date** et sa **description** (message des commits)

On quitte le log avec la touche **q**.

Pour retrouver qui a fait une modif sur une ligne précise du code, **git log** n'est pas le plus adapté (surtout si bcp de commits) :

```
git blame FILE
```

Cette commande liste toute modification faite sur FILE ligne par ligne. À chq modif est associé le début du **SHA** du commit où s'est passé la modif. De là, 2 possibilités :

1. Soit on fait un **git log** et on recherche le commit dont le **SHA** nous a interpellé via **git blame**,
2. Soit on se fait renvoyer directement les détails du commit avec

```
git show DEBUT_SHA
```

SE POSITIONNER SUR UN COMMIT PRÉCIS

Pour se positionner sur un commit (à la recherche d'erreurs)

```
git checkout SHA
```

Pour revenir à la branche principale (par défaut le commit le plus récent)

```
git checkout master
```

On ne peut pas supprimer un commit. Par contre on peut effectuer un commit **inverse** afin d'atténuer les effets produits dans le commit où on s'est trompé

```
git revert SHA
```

Pour modifier le message de son dernier commit (le commit ne doit pas avoir été **pushé** sur l'origine)

```
git commit --amend –m « Nouveau message du dernier commit »
```

Pour annuler tous les changements qu'on n'a pas encore commités

```
git reset --hard
```

LES REMOTES

Il est important d'avoir un **backup** de son code sur une autre machine, au cas où la notre tombe en panne par exemple.

Une fois travail fait et commits réalisés, on les envoie sur un remote. Un remote peut être interne (si on a plusieurs ordis par ex) ou externe (grâce à des services comme Github ou BitBucket). Utiliser un remote externe permet de travailler à plusieurs sur un projet. On peut visualiser ainsi (et agir sur) les serveurs distants qu'on a enregistré.

```
git remote
```

Si on a cloné un dépôt, on devrait au moins voir apparaître « origin » en réponse à la commande (nom par défaut donné par Git au serveur à partir duquel on a cloné le dépôt). *Faire attention si ssh ou https !*

Pour obtenir l'url complète du remote (origin est l'alias de l'url) :

```
git remote -v
```

On peut ajouter un remote (dépôt distant) :

```
git remote add [alias_remote] [url_remote]
```

On peut renommer un remote :

```
git remote rename [ancien_alias] [nouvel_alias]
```

Pour supprimer un remote :

```
git remote rm [alias_remote]
```

Voir https://git-scm.com/book/fr/v2/Les-bases-de-Git-Travailler-avec-des-d%C3%A9p%C3%B4ts-distants pour plus d'infos.

PUSH TO MULTIPLE REMOTES

```
# Create a new remote called "all" with the URL of the primary repo. Création d'un nouveau remote d'alias « all » avec l'url du repo principal
git remote add all [REMOTE_URL1]
# Re-register the remote as a push URL. On configure la 1ere adresse de push du remote « all » avec l'url du repo principal
git remote set-url --add --push all [REMOTE_URL1]
# Add a push URL to a remote. This means that "git push" will also push to this git URL. Puis on ajoute les autres url d'où on doit pusher vers le remote « all »
git remote set-url --add --push all [REMOTE_URL2]
```

On n'est pas obligé de créer un remote « all », on peut très bien se servir du remote « origin ». Dans ce cas, ne pas faire la 1ere instruction (git remote add all [REMOTE_URL1], mais directement à partir de la 2E instruction. Attention à l'option –push, pour fetch faire –fetch.

On peut aussi aller directement dans le **.git/config** du repo, dans la section **[remote REMOTE NAME]** pour ajouter, supprimer, modifier un remote.

On peut désormais pusher sur plusieurs remotes un repo en une seule instruction push

```
# Replace BRANCH with the name of the branch you want to push.
git push all BRANCH
```

RECUPÉRER DU CODE D'UN REPOSITORY

On peut copier un repo depuis GitHub sur sa machine. On utilise l'option **clone URL** du site, en bas à droite du repo. On colle le lien dans la commande suivante (touche Ins). Lorsqu'on clone un repo, le remote est automatiquement ajouté sous le nom **origin**.

```
git clone URL_REPO_GITHUB
```

Attention, il est assez compliqué de ne cloner qu'un seul fichier/répertoire d'un repo, car git fonctionne sur les repos.

Une solution ici : https://stackoverflow.com/questions/7106012/download-a-single-folder-or-directory-from-a-github-repo

Ou une autre :

```
git fetch
git checkout -m <branch> <yourfilepath>
git add <yourfilepath>
git commit
```

ENVOYER SON CODE SUR GITHUB

Pour synchroniser les modifs faites sur repo machine avec un repo GitHub, depuis la console, on se positionne dans son repo local. On fait un commit des dernières modifications, puis on **pushe** le repo vers GitHub, *c-à-d qu'on transfère depuis son espace local vers le remote choisi* (ici GitHub, mais ça pourrait être un autre ordi) *la situation de notre repo.*

On synchronise l'état de notre repo local vers l'espace où on le stocke

```
git push [remote] [branche]
git push origin master
```

On envoie nos modifs dans la branche **master** de notre remote.

La branche **master** est la branche qui contient le code courant de notre repo GitHub. Le remote sur lequel on envoie le code est appelé **origin** par défaut. Ici ce remote est GitHub, mais on pourrait en avoir un autre, ou même plusieurs, sur lesquels on enverrait le code.

La commande **git push** peut nous demander de nous **identifier** lors de la commande de transfert de fichiers, renseigner alors ses identifiants GitHub.

RÉCUPÉRER DES MODIFICATIONS

Pour récupérer les mises à jour du code (différent de cloner) lors d'un travail collaboratif par exemple :

```
git pull origin master
git pull [remote] [branche]
```

On envoie le code du remote vers le répertoire local. Il s'agit de l'inverse de la commande git push. **git pull** est la contraction des commandes **git fetch + git merge**. Parfois il n'est pas possible de se servir du git pull (ex : tirer sur plusieurs remotes en même tps), il faudra fetcher puis merger.

Penser à synchroniser son travail régulièrement, au risque de travailler sur une version du projet obsolète.

CRÉER DES BRANCHES

Les branches permettent de travailler sur des versions de code qui divergent de la branche principale (**master**). À l'initialisation d'un repo, le code est placé dans la branche master par défaut.

Pour voir toutes les branches du repo (**l'étoile devant une branche indique la branche de travail**) :

```
git branch
```

Pour créer une nouvelle branche

```
git branch [nouvelle_branche]
```

Pour se placer sur une autre branche du repo

```
git checkout [autre_branche]
```

Pour se placer sur une autre branche du repo

```
git checkout [autre_branche]
```

Pour supprimer une branche

```
git branch -d [branche]
```

FUSIONNER LES BRANCHES

Par exemple, pour ajouter dans la branche master ce sur quoi on a travaillé dans une branche divergente (cela fonctionne avec n'importe quelle autre branche que master et divergente)

```
git checkout master
git merge divergente
```

RÉSOUDRE UN CONFLIT

Il arrive très souvent d'avoir des conflits entre deux branches d'un projet qui empêchent de les fusionner (ex : 2 personnes travaillent en même tps sur un même fichier, modifiant la même ligne mais de façon différente). Git est en mesure de reconnaître ces différences et va nous indiquer en console pq il n'a pas pu merger les branches.

Il conviendra alors d'éditer le fichier en question

```
vim FILE_CONFLICTUEL
```

Les différences de contenu entre les deux branches s'afficheront à l'édition et il faudra choisir le contenu à conserver puis sauvegarder le fichier. Lorsque le conflit est résolu :

```
git status
```

De là, git nous indiquera « **unmerged paths** ». On pourra alors valider le merge non terminé par un commit sans message. Git détectera la résolution du conflit et nous proposera un message de commit, que l'on pourra modifier. On valide avec **:x** (commande vim). On obtiendra alors la confirmation que le merge s'est bien déroulé.

On pourrait aussi comparer les différences de versions avec les outils : *vimdiff*, *meld*, *opendiff*, *kdiff3*, *tkdiff*, *xxdiff*, *tortoisemerge*, *gvimdiff*, *diffuse*, *ecmerge*,*p4merge*, *araxis*, *emerge*. Pour lancer ces outils depuis git :

```
git mergetool
```

Ou

```
git vimdiff
```

IGNORER DES FICHIERS

Pour des raisons de sécurité et/ou de clarté, il est important d'ignorer certains fichiers quand on `git add` (ajoute à l'index), comme les fichiers de configuration (config.xml, database.xml, .env…), les dossiers et fichiers temporaires , les fichiers inutiles comme ceux crées par son IDE ou OS (.DS_Store, .project…)

Le plus crucial est de ne jamais versionner une variable de configuration (mdp, clé secrète d'API etc.), car cela conduit à une large faille de sécurité. **CHANGER LA VALEUR DES DONNÉES QUI AURONT ÉTÉ COMPROMISES ! (PUSHÉES SUR GITHUB PAR EX)**.

On déplace ces variables dans un fichier de configuration et on spécifie à Git qu'il faut l'ignorer : on liste dans le fichier `.gitignore` les fichiers à ne pas prendre en compte lors des versionnages du projet, **ligne par ligne**, en donnant **l'url complète de chaque fichier**. On crée un .gitignore pour chaque repo où il y aura des fichiers à ne pas tracker.

> **Il est impératif d'ajouter le .gitignore à l'index et de le commiter, sans quoi les informations ne seront pas prises en compte.**

EVITER LES COMMITS SUPERFLUS

Cas de figure : on est en train de travailler sur une fonction, quand on nous demande de nous plonger en urgence dans une autre branche voire repo.

Si on fait un commit de notre fonction alors que la modification n'est pas significative, cela va alourdir notre historique inutilement. Pour ne pas perdre les modifications en cours et passer à la tâche urgente, on peut mettre de côté nos modifs avec

```
git stash
```

On peut alors aller s'occuper de l'urgence en mettant notre fonction en **stand-by**. Une fois l'urgence traitée, on revient sur notre travail avec

```
git stash pop
```

Attention, `pop` libère le `stash` de ce qu'on y avait mis dedans. Soit on finit sa tâche, soit si on a encore besoin de la mettre de côté, on **re-stash** la tâche en cours.

Pour garder les modifications dans son stash

```
git stash apply
```

CONTRIBUER À DES PROJETS

Pour proposer une modification à un projet hébergé sur GitHub, on fait une **pull request** (PR).

D'abord, on regarde dans la doc du projet si des recommandations sont données pour faire une PR : certains demande un format de message de commit et ou de PR bien précis, d'ajouter des tests, etc. En général, on trouve cela dans le **README** du projet, sous le chapitre « *Contributing* » ou « *Pull Requests* »

Ensuite on récupère le repo sur lequel on souhaite contribuer : on **fork** puis on **clone** le repo. Forker permet de créer une copie du repo sur son compte GitHub ; sur la page du projet on clique en haut à droite sur **fork**, ensuite on **clone** l'url puis on **git clone [URL_CLONE]** en local.

Avant de commencer les modifs, on **crée une nouvelle branche**

```
git checkout -b [NEW_BRANCH]
```

On peut alors effectuer les actions qu'on désire. On n'oublie pas de **commiter** avec des messages clairs. Puis on **push**

```
git push origin [NEW_BRANCH]
```

Pas de push sur la branche master (ce sera au propriétaire du projet de merger notre branche sur son master, si les modifs lui conviennent).

Enfin, une fois les modifs transférées sur notre **fork GitHub**, on peut faire un **pull request**. Sur GitHub, dans notre nouvelle branche, cliquer sur **Compare & pull request** à droite. On sera amené à rédiger un message pour présenter sa proposition de modifications à l'auteur du projet. On recevra une notif qui nous indiquera si l'auteur prend ou non en compte notre proposition. L'auteur peut nous contacter s'il a besoin de précisions sur notre **PR**.

OBTENIR DE L'AIDE EN CLI

```
git --help # Aide sur les commandes génrales de git
git <command> --help # Aide sur une commande précise de git
```

CONNEXION EN SSH À GITHUB

En https, une adresse git est de cette forme :
https://github.com/USERNAME/REPO.git

L'avantage du SSH, c'est qu'on n'a pas à s'authentifier à chaque interaction avec le serveur git, puisqu'on passe par une clé rsa

Procédure https décrite sur git pour WSL.

Change remote URL'S https to SSH :

```
git remote set-url origin git@github.com:USERNAME/REPO.git
```

Verify that the remote URL has changed :

```
git remote -v
origin git@github.com:USERNAME/REPO.git (push)
```

With SSH keys, you can connect to GITHUB without supplying username & password at each visit (if not used during one year, it is removed).

1. Check for existing SSH keys **ls -la ~/.ssh**
2. If doesn't exist or don't want to use existing one, generate SSh key & ad dit to the SSH agent after creating the ssh key directory

```
mkdir .ssh && chmod 700 .ssh
```

```
ssh-keygen –t rsa –b 4096 –C « USERNAME@github.com »
```

When prompted « Enter a file in which to save the key », press Enter (accepts default file location) or type **/home/USR/.ssh/id_rsa** (or whatever)

```
chmod 600 .ssh/id_rsa && chmod 644 .ss/id_rsa.pub
# Faire une copie des clés dans Windows si un jour on désintalle WSL
/mnt/C/Users/user
```

After, it asks for adding a passphrase (Enter = no one ! Be careful, it is better to have one). For changing passphrase

```
ssh-keygen –p
```

After, start the ssh-agent in bg, it will return the agent PID :
(Attention, il faut refaire cette action à chaque fois qu'on ferme sa console WSL, eval + ssh-add)

```
eval « $(ssh-agent –s) »
```

Add private key to the ssh agent. Replace id_rsa if private key's directory has another name :

```
ssh-add ~/.ssh/id_rsa
```

3. Add public key to GH account, open public key file with editor & take it. In **upper-right corner** of any page (GH) click **profile photo > Settings**. In the user's side-bar select **SSH&GPG Keys**. Click **new ssh key** or **Add SSH key**. **Title field** = descriptive label. Paste key int **key field** and click **add ssh key**. If prompted, add GH password.
4. Test ssh connection :

```
ssh -T git@github.com
```

warning continue : type Y -> if all OK it prints « Hi username… »