Coding Weeks - Let's Not Panic

By Majora, et avec le soutien de la fabuleuse équipe de la cw11
Merci à tous les 1A pour vos questions qui sont moins débiles que ce que vous pensez

Cheatsheet Bash

On utilise une interface appelée shell pour communiquer entre l'utilisateur et le système d'exploitation. Bash est un shell très répandu. L'utilisateur interagit avec le shell via une CLI (Command Line Interface, ou Interface en Ligne de Commande), c'est-à-dire en tapant des lignes de code qui contiennent les commandes que le système d'exploitation doit exécuter. Un terminal est un programme qui ouvre une fenêtre graphique dans laquelle les utilisateurs peuvent taper ces commandes.

Ouvrir un shell

- VSCode contient un terminal intégré (View > Terminal), qui donne accès au shell Bash.
- Windows permet d'accéder au shell cmd, ou au shell PowerShell
- Sous Mac, c'est Terminal, pour accéder au shell Bash
- Sous Ubuntu, le Terminal permet de lancer le shell Bash
- Si vous avez une distribution plus exotique, vous n'avez probablement pas besoin de cette section du tuto

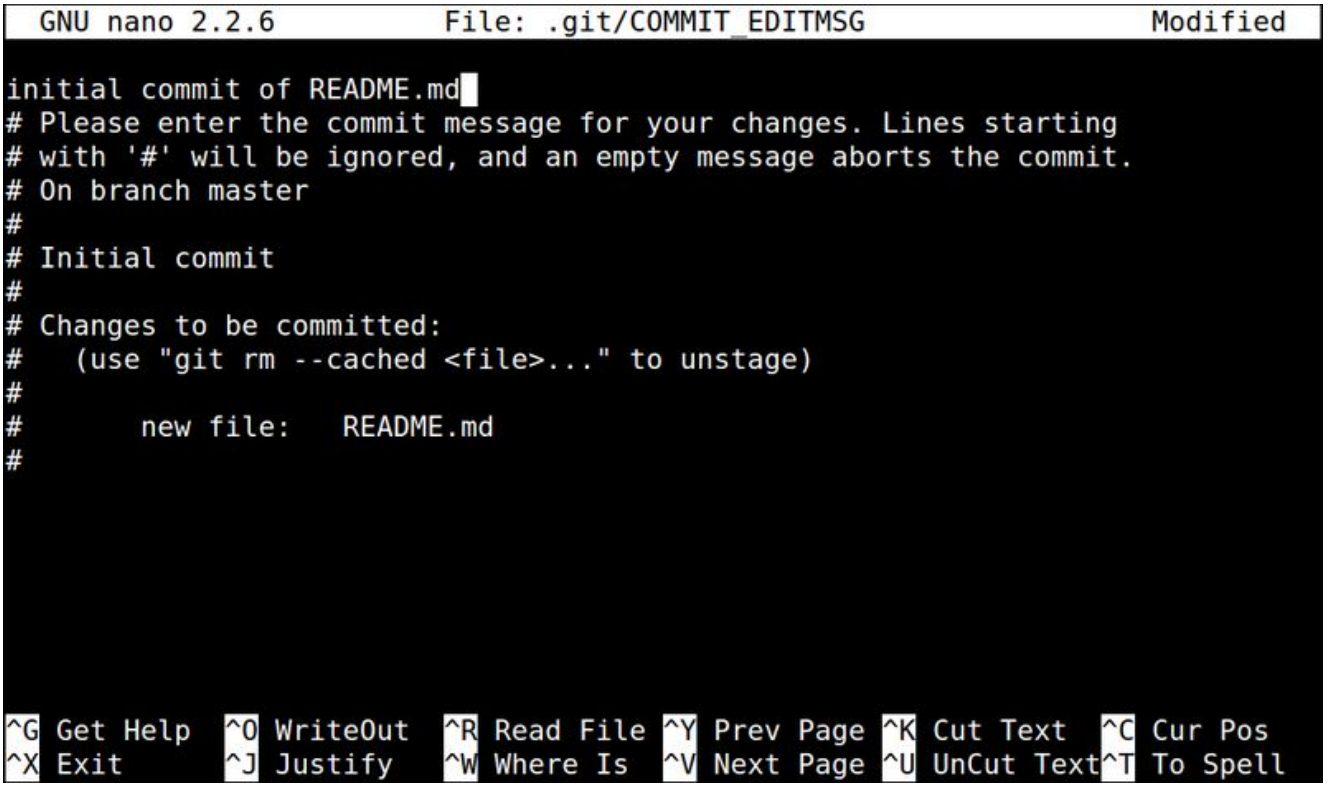
Commandes

Tous ces terminaux ouvrent des shells différents ; chaque shell a des différences subtiles dans les commandes qu'ils sont capables d'interpréter. Néanmoins, les commandes suivantes sont utilisables dans n'importe quel terminal :

- `pwd` : donne le nom du dossier parent. Utile pour comprendre où vous avez foutu votre dossier de travail.
- `cd folder_name` : "change directory". Permet de se déplacer dans l'arborescence de fichiers.
- Exemple : `cd Documents` pour rentrer dans le dossier "Documents" ; `cd ..` pour remonter d'un cran dans l'arborescence (retourner au dossier parent)
- `ls` : afficher les dossiers et fichiers du répertoire courant. Utile pour vérifier pourquoi votre autocomplete ne marche pas, ou si vous avez correctement réussi à bouger ou créer des fichiers
- `mkdir folder_name` : créer un dossier
- `touch file_name` : créer un fichier
- `mv path/towards/folder_name new/path/please/folder_name` : déplacer un dossier. Utile si vous avez lancé votre clone depuis

Aled j'ai ouvert nano

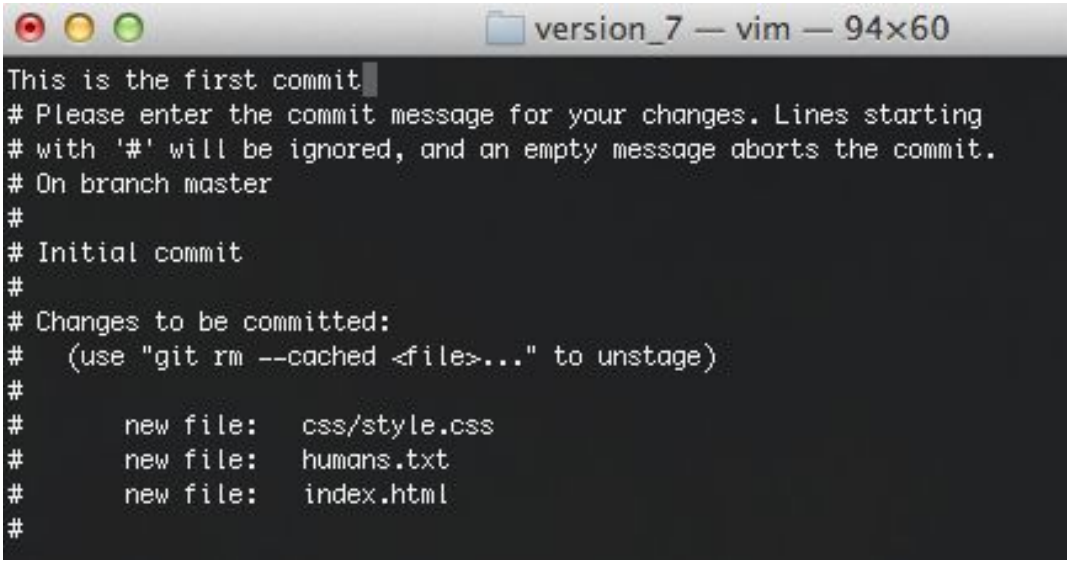
Vim est un éditeur en ligne de commande. On le reconnaît souvent au mot-clé "vim" dans le header (voir image), et aux aides de navigation en bas de l'écran (contrairement à vim, voir image plus bas). Le symbole `^` correspond à la touche `Ctrl` : `^O` signifie `Ctrl+o`. Les commandes de navigation sont donc données à l'aide de la touche Ctrl.



- `ctrl+o` pour quitter en sauvegardant
- `ctrl+x` pour quitter, s'il y a des changements nano demandera si on veut sauvegarder

Aled j'ai ouvert vim

Vim est un éditeur en ligne de commande. On le reconnaît souvent au mot-clé "vim" dans le header (voir image), et à l'absence des aides de navigation en bas de l'écran (contrairement à nano, voir image plus haut). La plupart des commandes sont tapées directement, pas besoin de la touche 'entrée' pour les valider.



- `i` : entrer en mode insertion (pour écrire du texte). Il y aura marqué "insert" en bas.
- `esc` : sortir du mode insertion
- `:x` ou `:wq` pour quitter en sauvegardant
- `:q` pour quitter sans sauvegarder

Pour le terminal de Mac, ou celui de VSCode : la touche tab permet d'auto-completer le nom d'un dossier ou fichier (s'il n'y a pas d'ambiguité possible : si les dossiers "Documents" et "Downloads" existent, il

faudra taper au moins "Doc" pour que le tab auto-complète avec "Documents"). L'autocomplete est sensible à la casse. Si ça ne marche pas, vous avez probablement une erreur de frappe.

Travailler ensemble sur du code

On utilise un outil de version control (gestionnaire de version) comme Git (voir Wtf is Git), de pair avec un service de hosting pour ce version control (hébergement en ligne des fichiers) comme GitLab ou GitHub.
Un projet bien construit contient un (ou plusieurs) fichiers README (voir Faire un bon README).

Wtf is Git

Git est un outil très utile pour coder en groupe. Il fait office de

- Communication avec un repo central : le repo (dossier) central est distant (hébergé en ligne), souvent sur un site comme GitHub ou GitLab. Le repo central distant sert de référence pour tous les développeurs : quand on veut modifier des fichiers dans le projet, on pull (récupère) le repo distant (qui est le plus à jour), on fait ses modifications, puis on push (envoie) vers le repo distant pour le tenir à jour.
- Gestionnaire de version (on peut consulter l'historique de version des différents codes de chaque personne qui contribue au projet). Exemple : Alice veut changer le schéma de couleurs de son jeu 2048. Elle se dit que des nuances de bleu, c'est pas mal. Elle change en bleu, fait un commit avec comme message "now in blue (first test)" et push vers le repo distant. Après discussion avec son client, il voulait plutôt du vert. Elle change donc en vert, commit avec "now in green after client feedback", et push vers le repo distant. Elle aura toujours accès aux versions précédentes, si elle a besoin de revenir en arrière, ou d'expliquer à Bob le stagiaire qui vient d'arriver pourquoi on est en vert et pas bleu : l'explication de chaque version est dans le message du commit.
- Merger intelligent : Git est capable de faire une fusion automatique entre les fichiers, en gardant les bonnes infos (la plupart du temps). Exemple : Alice pull le repo distant pour changer le schéma de couleurs de son jeu 2048. Bob pull le repo distant pour changer la police de caractère utilisée dans le jeu. Alice finit avant Bob (trop forte) et push son travail sur le repo distant. Bob finit un peu plus tard, se rend compte qu'il n'a pas la dernière version (celle d'Alice) et fait un nouveau pull. Git s'occupe d'intégrer le code d'Alice dans le repo local de Bob sans casser le code de Bob. Il est satisfait, et peut push, mettant le repo distant à jour.
- Branching : Git propose un système de branches, ce qui permet de travailler sur des features sans se marcher sur les pieds. Ex : Alice et Charlie veulent changer les couleurs du jeu. Bob travaille toujours en parallèle sur la police de caractère. Alice et Charles ont besoin d'échanger régulièrement et de se débeuguer l'un l'autre ; l'écran d'Alice n'affiche que du bleu, et Charlie est daltonien. Pour ne pas mettre sur la branche "master" qui est la référence, et ne pas polluer le travail de Bob avec des allers retours sur les teintes de rouge, ils vont travailler sur une nouvelle branche, qu'ils nomment colorBranch. Alice se place sur la branche colorBranch (en local) et travaille. Quand elle push, ce sera bien vers le repo distant, mais sur la branche distante colorBranch (pas master). Charlie pourra ainsi pull depuis le repo distant de la branche colorBranch, vers sa propre branche colorBranch locale. Quand il aura fini, il push vers colorBranch (du repo distant) etc. Tant qu'ils débeuguent encore, Bob ne verra pas leur travail sur la branche master, et ne sera pas pollué par leurs " #TODO" et "#ICI ÇA NE MARCHE PAS". Quand ils ont fini de débeuguer, Alice nettoiera le code, puis ouvrira une Merge Request (demande de fusion) de la branche distante colorBranch dans le branche distante master. GitLab s'occupe alors de fusionner automatiquement les 2 branches, ce qui incorpore le travail d'Alice et Charlie dans la branche master, sans casser le travail de Bob.

Faire un bon README

"README.md" est un fichier qui contient un texte explicatif sur le projet (voir https://www.makeareadme.com/). Le contenu est variable ; "Hello World!" est askip très populaire pour un commit initial, mais une fois que le projet a démarré, il vaut mieux commencer à expliquer le contenu du code.

- Pour les coding weeks, le README devrait contenir au moins :
    - Le nom de chacun des élèves du groupe
        - Entraînez-vous à utiliser Git en vous occupant chacun de votre propre nom ; voir Initial commit ou apprendre vos commandes git préférées
    - Un rappel de votre sujet
    - Une petite explication de votre structure de fichiers : où est le code à exécuter / le main ? Avez-vous un fichier de tests ? D'images ? Comment avez-vous organisé le code ?
    - Les instructions à taper pour faire tourner le code (pour mettre vos correcteurs dans une bonne disposition, et leur épargner 5-10 minutes de recherche dans votre code pour trouver tous seuls)
    - Les modules à importer si besoin
- Pour créer un fichier, on peut le créer via l'explorateur de fichier de l'ordinateur, via le bouton dans l'explorateur de fichier de VSCode (une fois qu'on a ouvert le dossier parent), ou via la ligne de commande avec "touch" (voir Cheatsheet bash).
    - Pour ouvrir un dossier dans VSCode, file > add folder to workspace
- le '.md' est une extension de fichier signifiant 'markdown', c'est du texte qui fait de la mise en forme automatique. GitLab a un interpréteur automatique de markdown intégré, mis en valeur dans le lien ci-dessus (https://www.makeareadme.com/), c'est assez usuel d'utiliser ce format sur GitLab et GitHub pour faire des readme. Si ça vous stresse vous pouvez utiliser le format "plain text" (README.txt) sans soucis.

Setup un repo Gitlab pour travailler ensemble

- Chaque personne se connecte au Gitlab de l'école :
- Si on choisit d'utiliser le protocole ssh (plutôt que https) : Chaque personne ajoute une clé ssh à son compte gitlab. Voir Gitlab avec une clé SSH
- Une seule personne crée le repo GitLab. Voir Créer un nouveau repo sur GitLab
- Cette même personne invite les autres membres du groupe. Voir Inviter les copains à travailler sur votre repo Git
- Une fois que tout le monde est maintainer, tout le monde clone le repo sur sa machine (voir Cloner un repo git)
- Tout le monde se place dans le repo fraîchement cloné (voir Cheatsheet bash)
- Une fois que tout le monde a un clone du repo en local (donc sur sa propre machine), tout le monde fait un commit initial (voir Initial commit ou apprendre vos commandes git préférées) avec un petit changement pour tester que le setup est bon.
- Le reste du tuto Git (les branches, etc…) est à priori dans l'exercice et les instructions. Bon courage ♡
    - Pour un tuto moins shitpost que le mien, essayez Atlassian ou la doc de GitLab qui apparaît quand vous créez un projet.

Connexion au Gitlab

Connexion à Gitlab : Il faut le lien vers le domaine GitLab que vous allez utiliser. Pour les Coding Weeks 2020, c'est gitlab-cw1.centralesupelec.fr
Choisir l'onglet "LDAP CS"
Identifiant : mail cs (c'est l'identifiant long pour les services de l'école)
Mot de passe : votre mdp du CAS CS (c'est probablement votre mot de passe Géode ou Edunao)

Gitlab avec une clé SSH

Pourquoi ? Pour éviter de devoir mettre ses identifiants à chaque push / pull comme c'est le cas avec https.
Comment ? En utilisant le terminal (voir Cheatsheet bash).

Générer une clé ssh

Dans un terminal :
`ssh-keygen -t ed25519 -C "<adresse mail>"` , cela génère une clé ssh chiffrée avec ed25519, remplacez <adresse mail> par votre adresse cs - c'est juste pour se rappeler de la clé ssh :) Toujours faire entrée sur les choix proposés et laisser une passphrase vide.

- Vous avez le choix entre clé RSA ou clé ed25519. Si vous connaissez pas la différence, faites la ed25519. Si vous voulez faire la RSA, les commandes sont similaires, cherchez en ligne ;)
- Si vous modifiez les choix par défaut, la commande suivante pour récupérer votre clé publique ne marchera pas.
    - Si par miracle vous arrivez à récupérer votre clé quand même à l'aide du terminal, il y a de grandes chances que vous vous preniez une erreur de `permission denied (publickey)` (voir cette erreur) plus tard quand vous tentez de push ou pull. Si c'est le cas, le plus simple est de générer une nouvelle clé sans changer les choix par défaut.
        - Si sur un malentendu vous réparez ça autrement, vous n'avez probablement pas besoin de ce tuto ;)
- Si vous choisissez une passphrase non vide, il faudra s'en rappeler, et la renseigner à chaque fois que vous faites un push ou pull avec git. C'est relou, mais plus secure. Pour les coding weeks, c'est clairement pas nécessaire.

Copier sa clé ssh

Dans un terminal :

- Mac : `pbcopy < ~/.ssh/id_ed25519.pub`
- Windows : `cat ~/.ssh/id_ed25519.pub | clip` (utilisez powershell, l'invite de commande de bash est claquée au sol, pour l'ouvrir vous recherchez powershell dans l'explorateur)
- Si vous choisissez une autre méthode (cat, ouvrir dans un bloc notes, que sais-je), faites attention à bien copier la totalité du fichier, sans oublier le début en 'ed25519' ni la fin avec votre adresse mail si elle y est.

Ajouter votre clé ssh

Ajoutez votre clé ssh toute fraîchement copiée à votre compte gitlab. Cliquez sur votre petite icône en haut à droite, puis Settings, puis SSH Keys à gauche. Coller votre clé, mettez lui un petit nom pour votre en souvenir (pas besoin de date d'expiration). Cliquez sur Add key.

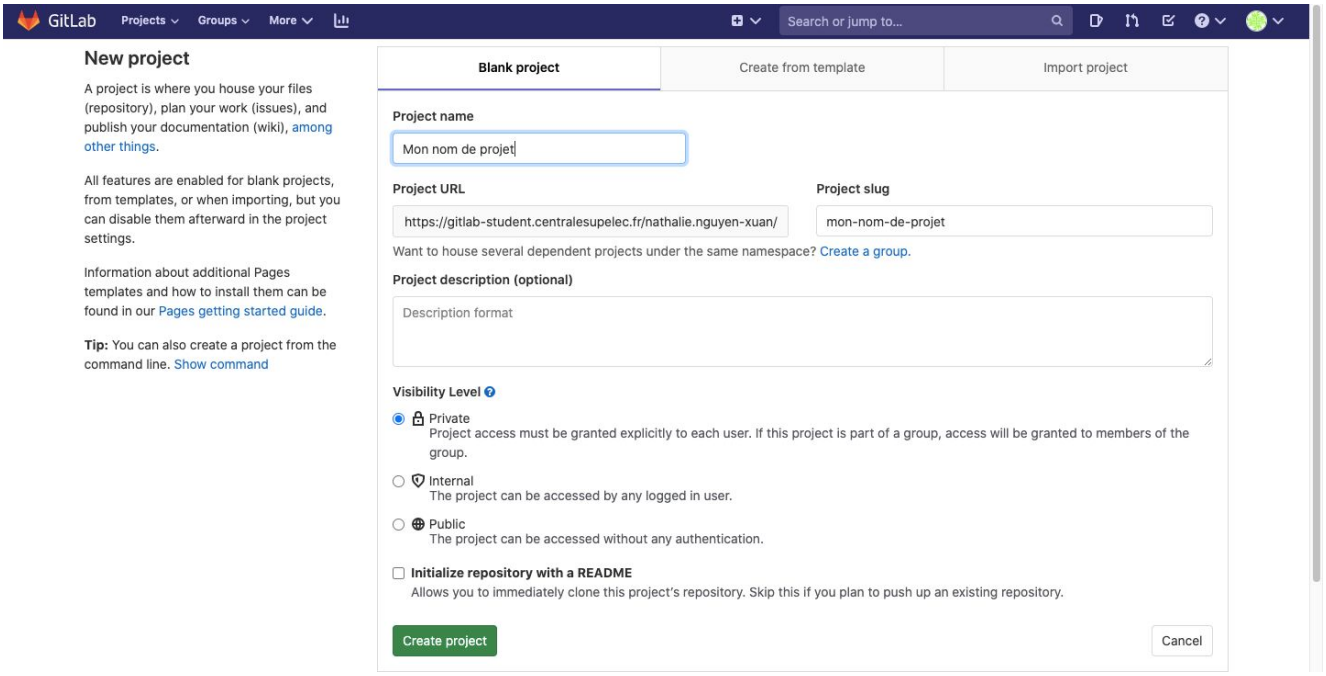
Pour vos projets suivants, inutile de refaire une nouvelle clé !

- Si vous utilisez le même domaine GitLab (dans le cas des Coding Weeks 2020, c'est gitlab-cw1.centralesupelec.fr), la clé ssh est déjà enregistrée dans votre profil (sous Settings > SSH Keys)
- Si vous utilisez un autre domaine, ou GitHub, il suffit de copier la clé que vous avez généré précédemment (reprendre l'étape "Copier sa clé ssh) et l'ajouter aux clés du nouveau domaine
- Pour faire plus avancé, il est possible de créer plusieurs clés avec des permissions différentes (pour contrôler notamment la sécurité), mais c'est franchement overkill pendant les coding weeks.

Si vous avez mis une date d'expiration, elle finira par expirer toute seule. Pour les Coding Weeks, c'est pas grave de laisser traîner des clés sur le GitLab de l'école.
Si vous codez plus sérieusement, avec un compte perso et des projets à vous, il vaut mieux faire attention à ne pas laisser traîner des clés, c'est potentiellement une faille de sécurité. C'est une bonne pratique de les renouveler régulièrement.

Créer un nouveau repo sur GitLab

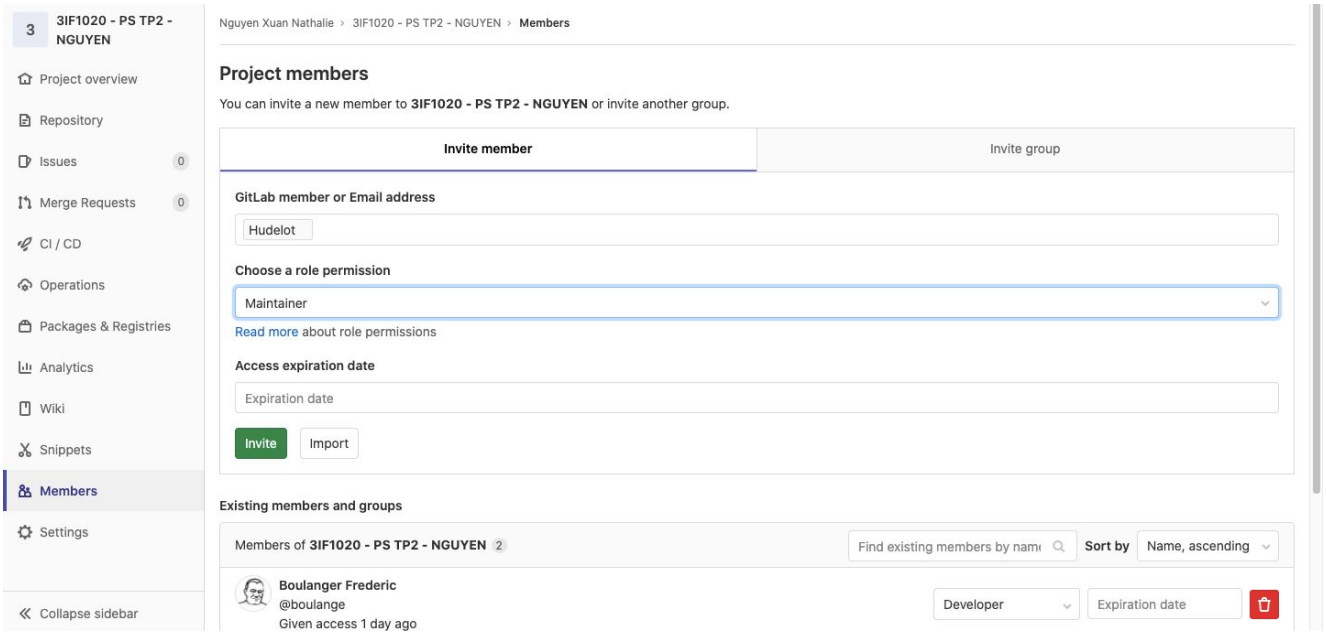
Sur la page d'accueil : "New Project"



Besoin de renseigner uniquement le nom du projet. Laisser l'url et le slug s'auto-remplir, la visibilité "private", et créer le projet.

Inviter les copains à travailler sur votre repo Git

Sur la page de votre projet (le mien s'appelle **3IF1020 - PS TP2 - NGUYEN**), dans l'onglet "Members" du menu, ajouter vos copains avec le rôle "Maintainer", et Céline Hudelot et vos référents de projet aussi tant qu'on y est.



Laissez la date d'expiration vide (ça serait dommage de se voir refuser l'accès à h-3 du rendu pour une erreur dans les dates).

Cloner un repo git

- Dans le terminal, se placer au préalable dans le dossier où vous voudriez voir votre nouveau repo (voir la commande cd dans Cheatsheet bash)
- Depuis GitLab, sur la page d'accueil de votre projet, copier l'URL de clonage par ssh (https c'est possible aussi, faudra juste renseigner des id / mdp)
- dans le terminal, `git clone <url>`

Si ça ne marche pas, c'est probablement une étape ratée lors de Gitlab avec une clé SSH

Initial commit ou apprendre vos commandes git préférées

Dans votre repo (=dossier) fraîchement cloné, en utilisant VSCode, créer un fichier "README.md" (voir Faire un bon README)

Git add

Dans le terminal : `git status`
Ça donne :

```
On branch master
Your branch is up to date with 'origin/master'.

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

      README.md

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

Le fichier README.md est "untracked" : il ne peut pas encore être commit.

Dans le terminal : `git add README.md`
Notre commande préférée est `git status` parce qu'on comprend r à ce qui se passe
Ça donne :

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file:   README.md
```

Git commit

Le fichier est prêt à être commit. Maintenant, la commande commit va commit tout ce qui est dans "changes to be committed", pas besoin de préciser les fichiers, mais ne pas oublier le commit message (signalé par l'option "-m" et un texte entre guillemets)
Dans le terminal : `git commit -m "Initial commit"`
Ça donne :

```
[master 474f9ec] Initial commit
 1 file changed, 1 insertions(+)
 create mode 100644 README.md
```

Note : si vous oubliez l'option `-m`, Git va ouvrir un éditeur en ligne de commande (probablement vim, ou nano) pour vous demander d'écrire un message. Vous avez 2 options, et dans les 2 cas vous aurez besoin d'interagir avec un éditeur en ligne de commande (voir vim et nano dans la Cheatsheet Bash) :

- écrire le message dans l'éditeur, quitter l'éditeur en sauvegardant. Le commit passera alors comme si vous aviez utilisé l'option `-m`
- quitter l'éditeur sans sauvegarder. Le commit sera avorté (pour cause d'absence de message ; recommencez la manip, avec l'option `-m` cette fois.

Notre commande préférée reste `git status` parce que sinon c'est assez opaque :

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

Git push

On est prêts à push, git nous l'a dit explicitement : `git push`
Résultat :

```
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master
```

nik.jpg

C'est normal, Git nous explique que notre projet en local (current branch, "l'aval") ne sait pas sur quelle branche du repo distant (l'origine, "l'amont", "upstream" en anglais) il doit push. Mais Git est très fort et pense avoir deviné ce qu'on tentait de faire, on va donc copier coller sa suggestion :

```
git push --set-upstream origin master
```

Résultat :

Si c'est toi qui a push en premier

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 289 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To ssh:repository
   42be914..ead1f82  master -> master
```

Si c'est pas toi

```
! [rejected]        master -> master (fetch first)
    error: failed to push some refs to '../remote/'
    hint: Updates were rejected because the remote contains work that you do
    hint: not have locally. This is usually caused by another repository pushing
    hint: to the same ref. You may want to first integrate the remote changes
    hint: (e.g., 'git pull ...') before pushing again.
    hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

niklewei.jpg

C'est dans le texte, il faut tenter un pull avant de pouvoir push, parce que tu n'es pas à jour avec le repo distant (origin). C'est parce que quelqu'un a push avant toi.
De manière générale, ***toujours toujours toujours pull avant de push***, histoire de pouvoir sereinement résoudre les merge conflicts.

Si il y a effectivement un merge conflict :

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

niklapolisse.jpg et passe sur la section Résoudre un merge conflict

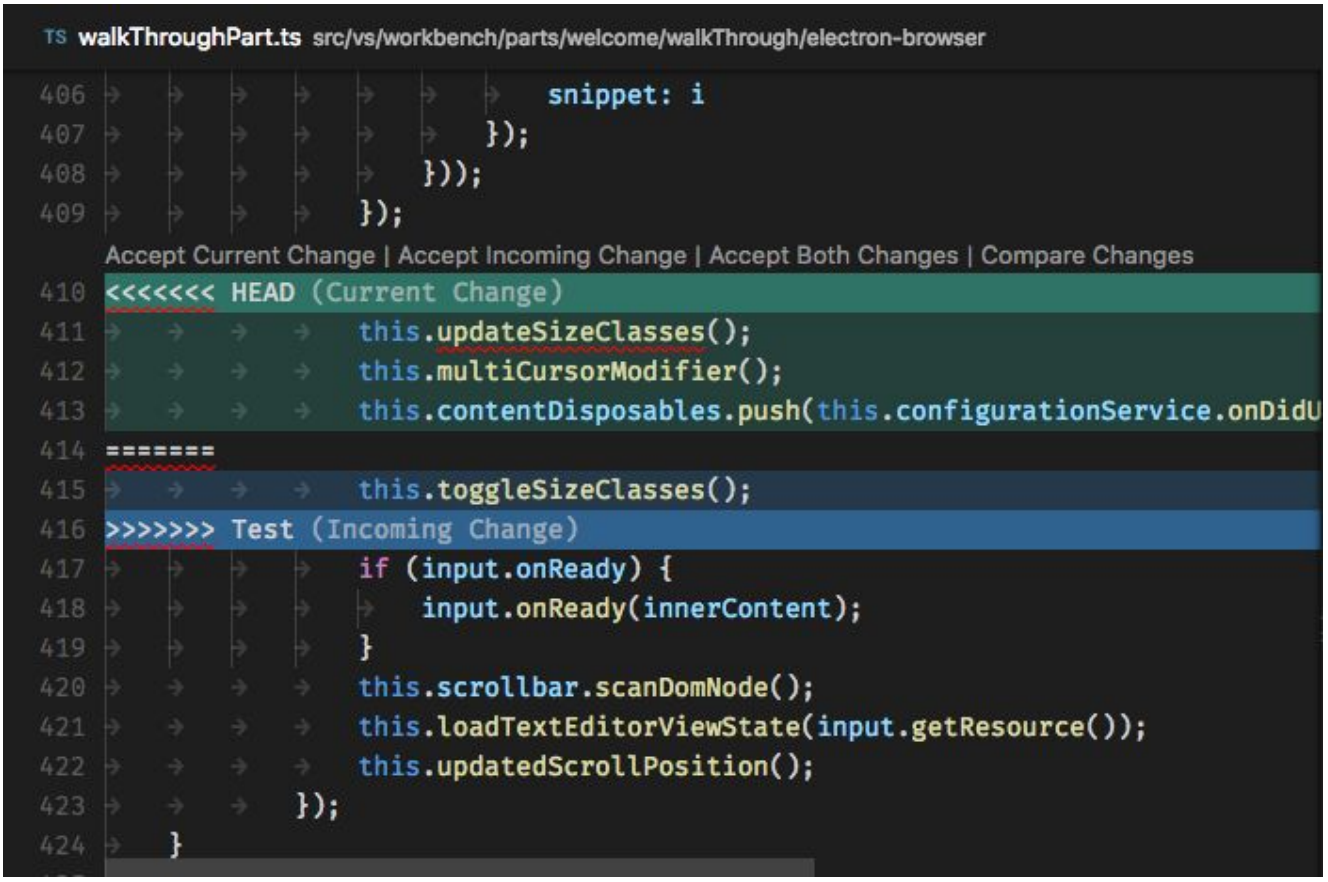
Sinon, félicitation, et `git status` :

```
On branch master
Your branch is ahead of 'origin/master' by 2 commit.
  (use "git push" to publish your local commits)
```

On est prêts à push pour de vrai cette fois.
Happy gitting !

Résoudre un merge conflict

VSCode a une interface très pratique pour résoudre les merge conflicts : les fichiers affectés (aussi listés dans le message d'erreur) sont affichés en bleu / violet dans l'explorateur de fichier de VSCode (panneau de gauche). Il suffit de cliquer dessus, puis pour chaque instance problématique, cliquer sur "Accept Current", "Incoming" ou "Both" selon le code que vous voulez garder. VSCode se charge alors de garder la bonne version et retirer tous les petits "<<<" relous.



Note : Vous avez aussi le droit de corriger à la main, dans ce cas ignorez le highlight automatique de VSCode, vous savez mieux que lui dtf.

FAQ

"Qu'est-ce que le "GitLab's instance domain" ?"

- C'est gitlab-cw1.centralesupelec.fr (l'instance privée sur GitLab, crée par l'école, pour les coding weeks)

"Je comprends pas où en est mon Git"

- `git status` car c'est notre commande préférée
- Lire attentivement les messages d'erreur : Git est assez doué en moyenne pour expliquer ce qui a raté

"Quand j'ai dû m'identifier après la commande git clone, j'ai rentré les mauvais id/mdp et depuis peu importe la manip, j'ai `access denied`. Comment modifier ce que j'ai rentrée avant ?"

- Tu as probablement fait un clone avec l'URL pour le protocole https (plutôt que ssh). Pas grave si ça te dérange pas de rentrer tes identifiants. Tu as probablement erroné ton mail ou ton nom d'utilisateur ; corrige-les avec
  `git config --global user.name "Nom d Utilisateur"` : probablement Nom Prénom, check sur ton profil GitLab. N'oublie pas les guillemets pour que la commande gère correctement les espaces entre ton nom et prénom.
  `git config --global user.email nomprenom@student-cs.fr`

"J'ai une erreur de type :

```
! [rejected]        master -> master (fetch first)
    error: failed to push some refs to '../remote/'
    hint: Updates were rejected because the remote contains work that you do
    hint: not have locally. This is usually caused by another repository pushing
    hint: to the same ref. You may want to first integrate the remote changes
    hint: (e.g., 'git pull ...') before pushing again.
    hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- C'est dans le texte, il faut tenter un pull avant de pouvoir push, parce que tu n'es pas à jour avec le repo distant (origin)

"J'ai une erreur"
Petit point astuce par votre srab Clément :

- Lire l'erreur. C'est tout bête mais il y a de grandes chances que le langage que vous soyez en train d'utiliser ait une gestion des erreurs correct et vous dise simplement ce qui ne lui a pas plu dans vos instructions, voir même vous propose des commandes à effectuer pour résoudre le problème.
- Copier coller l'erreur dans google : vous allez tomber sur des sites du genre stackoverflow ou des threads dans des issues github ou des gens qui auront eu le même problème que vous ont été aidés par d'autres gens et on trouve bien souvent la solution. Si personne n'a eu le même problème que vous (x doubt) vous pouvez même faire un post sur stackoverflow ou ouvrir une issue sur la librairie en question sur github par exemple.
- Essayez toujours de comprendre ce que vous faites quand vous utiliser des commandes : c'est super dur quand vous n'avez pas beaucoup d'expérience mais pas de soucis ça viendra. En debuggant posez vous la question de "que fait cette instruction et pourquoi elle est importante" ou "y avait-il des prérequis à l'exécution de cette instruction qui ont pu la faire foirer ?"
- Pour des aspects plus orientés code, refaites vous le cheminement complet de votre code ou de votre suite d'instruction, et même essayez de l'expliquer à un de vos potes. "J'ai utilisé cette commande pour installer cette librairie pour ensuite etc ...". Bien souvent vous vous rendez compte qu'effectivement vous avez oublié quelque chose ou fait un truc de travers.
- C'est assez compliqué on le sait très bien, aucun problème la dessus, et si vous êtes vraiment perdus et/ou dans l'incompréhension on vous aidera avec plaisir, mais si vous voulez essayer par vous même peut être qu'avec ça vous aurez plus de pistes !

"Nous on est en pls" ou "clairement je suis abattu" ou "Aled"

- Git c'est dur à appréhender, c'est normal
- On vous laisse pas tous seuls ♡
- Pingez un 3A ou un encadrant sur Teams pour qu'on vous débloque en vocal
- Oubliez pas de boire de l'eau et respirer

Python

Import de fichiers et modules, et exécution des scripts

Par Gianluca Quercini
Dans le cas où votre projet est organisé selon une arborescence de répertoires
Supposez que vous avez l'arborescence suivante dans votre projet:

- Un répertoire P qui contient tout votre projet.
- Un répertoire A dans P qui contient le fichier test.py
- Un répertoire B dans P qui contient le fichier main.py

Vous voulez importer dans `main.py` la fonction `my_test()`. Dans le fichier `main.py` vous devez écrire :

```
from A.test import my_test
```

Pour exécuter le fichier main.py, je vous suggère d'utiliser directement le terminal,
sinon si vous faites un click sur la flèche verte, cette exécution sera faite
relativement au répertoire où main.py se trouve (à savoir B), donc le import A.test ne sera pas trouvé.
Ouvrez le terminal, placez-vous dans le répertoire racine P du projet et tapez la commande suivante :

```
python -m B.main
```

De cette manière, on est sûr que Python utilisera le répertoire P comme répertoire de travail. Toute référence qu'il trouve (par ex., `from A.test import my_test`) sera interprétée relativement à ce répertoire.

Mes listes Python se modifient toutes seules

En Python, une liste (ou array, ou liste de liste…) est un objet qui a une scope globale (pas comme les float ou les int, par exemple). Si elle est modifiée dans une fonction enfant, elle le sera aussi dans la fonction parent, et ce peu importe le retour :

```
def fEnfant(l) :
      for i in range(len(l)):
            l[i] = l[i] * 2
      return l

def fParent() :
      liste = [1, 2, 3]
      print("liste originale : ", liste)
      deuxiemeListe = fEnfant(liste)
      print("liste après fEnfant : ", liste)
      print("liste retournée par fEnfant : ", deuxiemeListe)
      assert(liste[0] == 1)
```

Sortie :

```
liste originale : [1, 2, 3]
liste après fEnfant : [2, 4, 6]
liste retournée par fEnfant : [2, 4, 6]
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    fParent()
  File "main.py", line 12, in fParent
    assert(liste[0] == 1)
AssertionError
```

On a pas seulement retourné une liste de doubles, on a par erreur modifié la liste originale. C'est pour ça que c'est une bonne pratique de ne jamais modifier les arguments d'une fonction. Si on a besoin de modifier ou d'utiliser une liste qui est un argument, on en fait une copie

```
def fEnfant(l) :
copie = l.copy()
```

```
# ou bien
# copie = list(l)
      for i in range(len(copie)):
            copie[i] = copie[i] * 2
      return copie

def fParent() :
      liste = [1, 2, 3]
      print("liste originale : ", liste)
      deuxiemeListe = fEnfant(liste)
      print("liste après fEnfant : ", liste)
      print("liste retournée par fEnfant : ", deuxiemeListe)
      assert(liste[0] == 1)
```

Sortie :

```
liste originale : [1, 2, 3]
liste après fEnfant : [1, 2, 3]
liste retournée par fEnfant : [2, 4, 6]
```

Pas de message d'erreur, l'assertion est réussie !

Pytest ne marche pas

- As-tu installé Pytest dans le bon interpréteur ?
- As-tu bien Python dans ton PATH ?
- As-tu bien la même distribution (version) de Python sélectionnée lors de l'installation de Pytest, et sur VSCode / ton terminal ? Il faut vérifier que Pytest a bien été installée sur la distribution qu'utilise ton terminal.
- As-tu une commande de la forme :
  `pytest --cov=src --cov-report html tests/test_grid_2048.py`
  - Je veux lancer plusieurs tests :
    `pytest --cov=src --cov-report html tests/test_*.py` si tu es sous autre chose que Windows, utilise une "wildcard" (*) pour exécuter la commande sur tous les fichiers qui s'appellent "test_quelquechose.py"
  - Le wildcard ne marche pas : tu es probablement
    - dans le mauvais dossier : tes fichiers tests sont-ils dans un dossier test ? Es-tu toi-même dans le fichier parent de test ? Utilise `ls` (voir Cheatsheet bash) pour vérifier.
    - dans un shell Windows qui ne supporte pas les wildcards. Utilise
      `pytest --cov=src --cov-report html tests/`

Tkinter n'arrive pas à lire les inputs de mon clavier

- Ce topic pour une courte explication de l'intérêt de la méthode `focus_set()`
- Implémentation :

```
c = Canvas(root, width=400, height=400)
# [...]
c.focus_set()
root.mainloop()
```

Autres conseils

Extensions utiles pour VSCode

- GitGraph
- GitLens
- LiveShare

Python

- Guide pour les imports en Python
- Guide pour un peu de GitLab, mais aussi Python3 et Django