DEV25 - VEILLE TECH

FIJ - Sid Englebert - Mis à jour en mai 2025

Table des matières

I. Concepts élémentaires 4
1. Ordinateurs, serveurs, clients 4
2. Différents serveurs et hébergements 5
3. Langages côté serveur et côté client 6
4. Systèmes d'exploitation 7
a. Pourquoi utiliser Linux sous Windows ? 7
b. Installer et configurer le WSL 7
c. Connexion au Remote WSL 8
5. Stades et environnements de développement 8
6. Interfaces 9

II. Interfaces en ligne de commande 10
1. CLI 10
2. Shell 10
3. Émulateurs de terminal 10
4. Résumé 11
5. Commandes de base de CMD (Windows) 11
6. Programmes Windows utilisables via CMD 12
7. Commandes de base de Bash (Linux) 12

III. Contrôle de Versions 14
1. Intro 14
a. C'est quoi un VCS. Pourquoi Git ? 14
b. Les deux usages de Git (tracking & collaboration). 14
c. Comment utilise-t'on Git ? 14
2. Utilisation 14
3. Workflow classique de Git 15
4. Exercice : Premier Commit 17
5. Bonnes pratiques du commit 18
6. Add et commit en même temps 18
7. Supprimer un fichier à plusieurs étages 18
8. Déplacer un fichier à plusieurs étages 19
9. Ignorer des fichiers avec .gitignore 19
10. Les branches 20
a. Master vs. Main 20

b. Changer le nom de la branche par défaut 20
c. Renommer la branche principale d'un dépôt existant 21
d. Créer une nouvelle branch 21
e. Fusionner deux branches 21
f. Changer l'origine d'une branche divergente (rebase) 22

IV. Synchroniser nos projets avec Github 23
1. Git en ligne et Github 23
2. Sauvegarder et synchroniser ses projets 23
a. Cloner un repo distant 24
b. Push & Pull 24
c. En pratique, se connecter à notre premier repo 24
d. Tirer les changements 25
e. Vérifier les changements 25
f. Conclusion 26
3. Gestion des branches en remote 26

V. Divers 28
1. Rédiger des documentations avec Markdown 28

VI. Collaboration avec Github 30
1. Les différents workflows collaboratifs 30
a. Centralisé (SVN-style) 30
b. Feature Branch Workflow 30
c. Trunk Based Development (TBD) 30
d. Gitflow 31

I. Concepts élémentaires

1. Ordinateurs, serveurs, clients

Lorsque l'on parle de « Clients » et de « Serveurs » en informatique on fait référence à deux programmes qui communiquent entre eux. L'un fait des requêtes, l'autre y répond. L'exemple le plus courant est

- un serveur web (Apache ou Nginx) qui « sert » un site internet à des
- clients, les navigateurs des visiteurs du site.

N'importe quel ordinateur peut être un serveur, vous avez peut-être déjà utilisé des serveurs web locaux comme **Wamp** ou **Xampp**. En utilisant un tel programme on peut consulter un site qui est *servi* localement, habituellement à l'url **http://localhost** ou **http://127.0.0.1**. On pourrait également servir un site internet depuis son ordinateur personnel vers le web tout entier : il suffirait pour cela qu'on obtienne notre propre adresse IP, qu'on fasse les réglages nécessaires dans notre routeur et dans notre firewall. On peut également assigner un nom de domaine à cette adresse IP mais ce n'est pas absolument nécessaire.

Servir du contenu web depuis une machine locale pose tout un tas de problèmes qui sortent souvent des compétences strictes des web-développeur·euse·s.

- L'ordinateur doit rester allumé en permanence sinon le service hébergé est hors-ligne.
- Les risques de sécurité concernent désormais notre réseau local et notre machine personnelle.
- On doit nous même administrer le *stack* (une « pile », ensemble de programmes) logiciel. Choisir ce *stack*, le système d'exploitation, etc.
- On doit nous même créer les connexions entre notre adresse IP et le nom de domaine. Ce qui est particulièrement un problème en Belgique puisque les adresses IP non-business sont traditionnellement dynamiques plutôt que fixes, c'est à dire qu'elles changent régulièrement plutôt que d'être attribuées définitivement à chaque client.
- Et bien d'autres…

Ce sont quelques-unes des raisons pour lesquelles les web-développeur·euse·s louent des espaces d'hébergement à des sociétés spécialisées comme *OVH*, *Amazon AWS, Microsoft Azure*, etc. Les services web vivent alors dans des « Fermes à serveurs » les fameux data-centers.

Une réplique du premier serveur web, déployé en 1990 au CERN. Pour éviter qu'il ne soit accidentellement éteint, son concepteur Tim Berners-Lee avait écrit à la main à l'encre rouge : This machine is a server. DO NOT POWER IT DOWN!!

2. Différents serveurs et hébergements

- Hébergement **Web** : Un stack semblable à Wamp, c'est à dire Apache, MySQL, PHP. Destiné à l'hébergement web classique.
- Hébergement **Wordpress** : Ce type d'hébergement n'autorise qu'un seul type d'application à tourner, dans ce cas Wordpress. Il existe d'autres formules du même genre (avec des serveurs Minecraft par exemple). C'est donc un hébergement web censé être optimisé pour un seul type d'application.
- Hébergement **Shared** : Ou *mutualisé*. Les ressources sont mutualisées entre plusieurs développeurs, c'est souvent le cas pour l'hébergement web où les ressources demandées sont faibles. Les différent·e·s utilisateur·ice·s partagent la même adresse IP.
- Hébergement **Dedicated** : Ou *dédié*. Les ressources et une adresse IP unique sont garanties à chaque utilisateur·ice.
- Hébergement **VPS** : *Virtual Private Server*, c'est un hébergement dédié

au sein d'une machine virtuelle. Elle est utilisable intégralement. La plupart du temps nous pouvons ainsi y choisir le système d'exploitation, les programmes installés, etc.

- Hébergement **Bare-Metal** : C'est un hébergement dédié dans une machine physique intégralement louée. Ce type d'hébergement est assez cher.
- Hébergement **Managed** : Le gros de la maintenance logicielle est assuré par l'hébergeur mais la gestion est assurée via un panneau de contrôle facile à gérer, le plus populaire est CPanel.
- Hébergement **Unmanaged** : Tout est fait par l'utilisateur·ice, y compris redimensionner les partition, mettre à jour l'OS, mesurer les performances techniques, renouveler les certificats de sécurité SSL, gérer la communication des serveurs de noms de domaine, etc.
- Hébergement **Cloud** : L'hébergement cloud fait référence à un type d'hébergement décentralisé qui alloue des ressources dynamiquement, distribue les charges utilisateurs géographiquement, etc. Ce type d'hébergement n'est pas classiquement utilisé pour de simples sites internet même si c'est techniquement possible. Ce type d'hébergement est en général facturé à la consommation plutôt qu'au forfait. Notez qu'on désigne généralement par « cloud » l'ensemble des serveurs internet mais que ce terme sert également à désigner ce type de produits spécifique.

Chacune de ces options offrent des avantages et inconvénients en terme de coût, de temps de travail, de compétences nécessaires, de sécurité, etc.

La création de sites internet et la maintenance des serveurs sur lesquels ils vivent sont deux corps de métiers différents. On parlera de

- **DevOps** : Pour le développement des applications ;
- **SysOps** : Pour la maintenance des infrastructures système.

3. Langages côté serveur et côté client

Certains langages s'exécutent «côté serveur», d'autres «coté client». Dans le cas du web, comme on l'a dit plus tôt :

- Le serveur est constitué par un stack Apache, PHP, MySQL.
- Le client (le navigateur internet) affiche de l'HTML et du CSS, et exécute du Javascript (JS).

Lorsque les développeur·euse·s écrivent du PHP celui-ci n'est jamais « servi » directement au client. Il est exécuté sur le serveur pour fabriquer du contenu

HTML/CSS/JS, qui sera le seul accessible au client.

Ce principe permet de servir au client un contenu adapté à sa requête. Le HTML étant un type de code complètement statique, si on ne dépendait que de lui on serait obligé de fabriquer chaque page web à la main. Par exemple, lorsqu'on fait une recherche sur un site de e-commerce, si on tape « smartphone 8gb 5g gaming », le PHP côté serveur va « fabriquer » la page demandée et renvoyer le contenu nécessaire au client.

Un autre exemple est **FTP**. Lorsque vous aurez développé un site internet vous devrez transférer les fichiers de votre site sur un serveur en ligne. La façon la plus classique de procéder est d'utiliser le *File Transfer Protocol*. Dans cet exemple il y a donc un serveur FTP qui tourne sur l'hébergement (celui-ci a une adresse et demande un nom d'utilisateur et un mot de passe) et on utilise un client FTP comme **Filezilla** ou **WinSCP** pour se connecter à ce serveur depuis son ordinateur personnel.

4. Systèmes d'exploitation

a. Pourquoi utiliser Linux sous Windows ?

On a l'habitude de travailler sur des systèmes d'exploitation commerciaux comme Windows et MacOS. De leurs côtés les serveurs d'hébergement tournent dans leur immense majorité sous Linux. C'est pourquoi Windows a récemment développé le « Windows Sub-system for Linux », une machine virtuelle sous Linux, utilisable directement dans Windows. Ceci afin de permettre aux développeur·euse·s de travailler de la même façon interagissant avec leurs hébergements.

Il n'existe pas de versions « standard » de Linux, mais une multitude de « distributions » (ou *distros*) avec des différences difficiles à résumer dans ce cours.

b. Installer et configurer le WSL

Le WSL n'a pas d'interface graphique par défaut puisqu'on s'en sert en ligne de commande. Par défaut lorsqu'on installe le WSL sans spécifier de distribution particulière c'est Ubuntu qui sera installé par défaut, ce qui est parfait pour notre cas d'usage. D'autres distros sont disponibles comme Debian (la distro la plus standard), Kali (une distro orientée cybersécurité), ou d'autres.

```
> wsl --install (installer la distro par défaut)
> wsl --list (lister les distros installées)
```

Lors de l'installation le système va demander la création d'un nom d'utilisateur

et d'un mot de passe. Dans le cadre de ce cours afin de faciliter le debugging par la suite nous utiliserons tou·te·s le couple user password suivant:

```
user
password
```

c. Connexion au Remote WSL

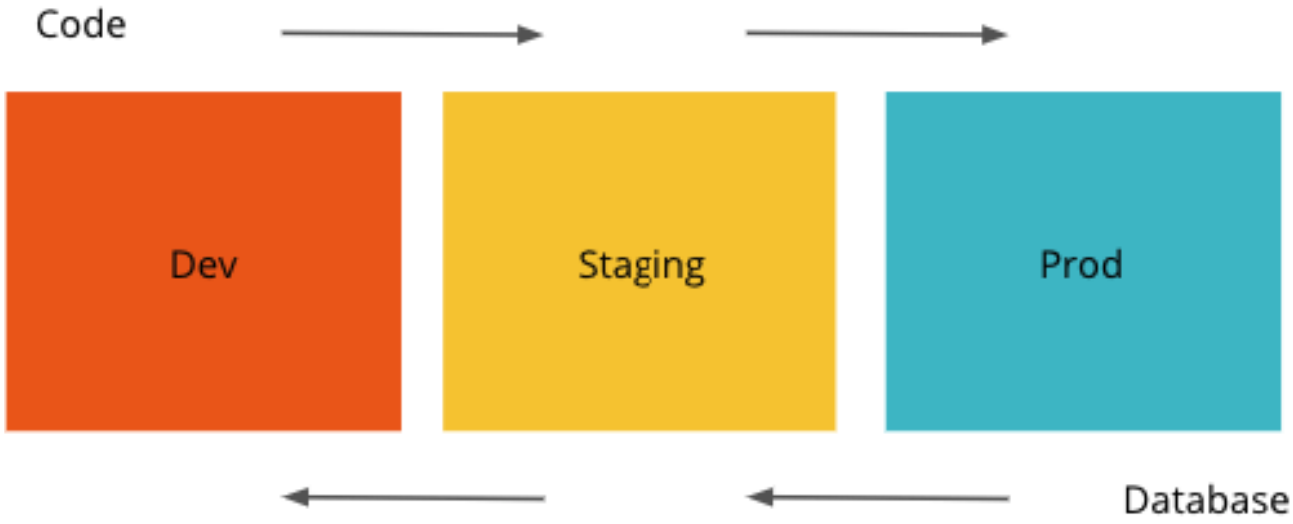
Durant les premiers cours, on pourra utiliser le terminal WSL directement avec le VS Code de Windows. Mais à partir de la partie « Collaboration », on utilisera « Connexion au Remote WSL » en utilisant ce logo en bas à gauche de VS Code.

En plus de meilleures performances, cela nous permettra de ne pas avoir de conflits au niveau du retour à la ligne, qui est géré différemment sur Windows et Linux/MacOS.

5. Stades et environnements de développement

Les stades de production :

- *Développement* (le code que vous êtes en train d'écrire/modifier).



- *Staging* (le code proposé au déploiement).
- *Production* (le code déployé en ligne).

À ce stade du cours il peut être difficile de voir pourquoi on passe par cette étape du *staging* mais cela prendra tout son sens lorsqu'on travaillera en ligne, voir le chapitre **collaboration**.

Si le code va toujours du bas (développement) vers le haut (production), les

données déjà déployées (la base de données, les images uploadées, les plug-ins, etc.) iront toujours du haut vers le bas. Nous y reviendrons quand on parlera de *.gitignore*.

6. Interfaces

La CLI et la GUI, sont les deux façons les plus courantes d'interagir avec un ordinateur, que ce soit localement ou à distance.

CLI : *Command Line Interface*. **Voir le chapitre consacré.**

GUI : *Graphical User Interface*. Généralement les interfaces graphiques sont des programmes qui permettent d'utiliser la CLI sans taper dans un terminal.

IDE : *Integrated Development Environment*. Ensemble d'outils dédiés au développement de code. *Sublime Text* n'est pas un IDE, c'est un éditeur de texte riche. On peut presque en faire un IDE en lui ajoutant des plug-ins. Mais philosophiquement, Sublime veut rester un éditeur de code minimaliste.

Par abus de langage on parle de certains éditeurs de code « riches » comme des IDEs. C'est le cas de *Visual Studio Code* et de *Sublime Text* qui se rapprochent des IDEs mais sont immensément plus légers et minimalistes que de véritables IDEs. La différence formelle entre un éditeur riche et un IDE est l'intégration de compilateurs. Les développeurs qui ne font que du web n'utilisent pas d'IDEs puisqu'ils n'utilisent habituellement pas de langages compilés.

Exemples d'outils inclus dans les RTE/IDE :

- Éditeur de code
- Coloration syntaxique selon le language de programmation choisi
- Navigateur de fichiers
- Debugger
- Compilateur (**Uniquement IDE**)
- Correction syntaxique
- Terminal
- Contrôle de version
- Remote (FTP, SSH, etc.)
- Co-pilote IA
- Serveur live intégré

II. Interfaces en ligne de commande

1. CLI

Command Line Interface. L'interface en ligne de commandes fait référence à toute façon d'interagir directement entre un utilisateur et un système d'exploitation (Windows, Linux).

2. Shell

Un Shell est un programme qui interprète et exécute les commandes de l'utilisateur. *Exemples : Bash, Zsh, Windows Command Prompt (CMD), Powershell.*

3. Émulateurs de terminal

Historiquement on utilisait des engins physiques pour interagir avec les ordinateurs *mainframe* (centraux), on les appelait des téléscripteurs puis des terminaux. Dans l'époque contemporaine, ces terminaux ne sont plus *hardware* mais *software*, on parle donc d'émulateurs de terminal comme on parlerait d'un émulateur d'une vieille console de jeux vidéo. Par abus de langage, le mot *terminal* remplace aujourd'hui la forme longue.

Dans l'utilisation moderne également, les web-développeurs hébergent leurs sites internet sur des serveurs, qui sont en fait des ordinateurs toujours allumés, sans écran et sans clavier connectés à internet. L'utilisation d'un terminal sur notre ordinateur local pour interagir avec notre serveur est un peu similaire avec l'utilisation historique des terminaux pour interagir avec le *mainframe*.

Ci-dessus, un téléscripteur T100 et un terminal VT100, au sens propre du terme, remplacés aujourd'hui par nos émulateurs.

Le terminal est la *fenêtre* dans laquelle on adresse des commandes au shell, qui

interagit avec le système d'exploitation.

Par un terminal on peut interagir avec plusieurs *shell* différents. Par exemple sous Windows, l'application *Windows Terminal* permet d'utiliser au choix n'importe quel shell (CMD, Powershell et même le Bash de Linux avec WSL). *Exemples : Windows Terminal, Gnome Terminal.*

4. Résumé

- Command Line Interface : Toute façon d'interagir en ligne de commande.
- Shell : Tout programme qui interprète des commandes.
- (Émulateur de) terminal : le programme dans lequel on tape les commandes.

5. Commandes de base de CMD (Windows)

```
> help (afficher les commandes par défaut)
> help dir (aide pour la commande xxx)
> dir (Afficher le dossier courant et son contenu)
> cd Desktop (Se déplacer vers le dossier Desktop)
> cd Downloads\Folder\Subfolder (Se déplacer vers...)
> cd .. (Se déplacer vers le dossier parent)
> cd ..\Desktop (Se déplacer vers le dossier frère)
> cd / (Se déplacer vers la racine du répertoire)
> mkdir Test (Créer le dossier Test)
> md Test (Équivalent à mkdir)
> echo hello > file.txt (Écrire dans un nouveau fichier)
> del file.txt (Effacer le fichier)
> rmdir Test (Effacer le dossier Test)
> rd (Équivalent à rmdir)
> mv file.txt file.css (Déplacer/renommer le fichier)
> echo world >> file.txt (Ajouter au fichier)
> clear (Nettoyer l'écran)
```

(Annuler la commande en cours)

(Coller, clic droit produit le même effet)

(Afficher la suggestion, si disponible)

 ou  *(Revoir les commandes précédentes)*

6. Programmes Windows utilisables via CMD

Si les programmes sont installés et ajoutés au PATH :

> explorer . *(Ouvrir l'explorateur de fichiers dans le dossier courant)*

> notepad file.txt *(Ouvrir le fichier dans le bloc-notes)*

> code . *(Ouvrir le répertoire courant dans Visual Studio Code)*

> winget search Package *(Rechercher «Package» dans Winget)*

7. Commandes de base de Bash (Linux)

Sous Linux, le shell standard est Bash. La plupart de ces commandes utilisent des options, on utilisera *man* pour les connaître. Les commandes les plus utilisées sont:

> man xxx *(afficher l'aide pour la commande xxx)*

> cd *(change directory)*

> ls *(lister le contenu)*

> touch file.txt *(créer le fichier)*

> mkdir *(créer le dossier)*

> grep 'text' file *(Rechercher 'text' dans le fichier spécifié)*

> pwd *(afficher le répertoire actuel)*

> mv file1 file2 *(renommer ou déplacer)*

> rmdir *(supprimer un dossier vide)*

> rm *(supprimer un dossier/fichier)*

> cat *(Court pour «concaténation», c'est une commande versatile qui permet plusieurs choses, afficher, fusionner, etc. Voir les options).*

> cat file.txt *(Afficher le contenu du fichier dans le terminal)*

> cat file1.txt file2.txt *(Afficher le contenu des deux fichiers dans le terminal)*

> cp file.txt newfile.txt *(copier)*

> history *(historique du terminal)*

> clear *(Nettoyer la fenêtre)*

> top *(Afficher les processus actifs)*

III. Contrôle de Versions

1. Intro

a. C'est quoi un VCS. Pourquoi Git ?

Pour gérer du code informatique, les développeurs ont besoin de pouvoir garder une historique et collaborer ensemble. Ce sont les deux usages principaux de Git, le logiciel le plus utilisé au monde dans ce domaine.

Git est un «Version Control System».

b. Les deux usages de Git (tracking & collaboration).

Git garde une historique des modifications du code dans une base de données spéciales appelées «Repository» (Dépôt, souvent abrégé « repo »). Il nous permet d'éviter de faire des back-ups complets d'un projet de développement. Ce qui serait long, complexe et provoquerait des catastrophes.

c. Comment utilise-t'on Git ?

- CLI
- Rich Text Editors (VS Code, Sublime)
- GUI (Gitkraken, Sourcetree).

2. Utilisation

Pour vérifier que l'on a Git accessible dans notre terminal :

```
> git --version
```

Configurer Git pour la première fois. On doit spécifier plusieurs choses lorsqu'on commence à utiliser Git :

- Nom
- E-mail
- Default editor
- Line Ending.

Ces paramètres peuvent être spécifiés à trois niveaux:

- System (Tous les utilisateurs de l'ordinateur).
- Global (Tous les repos de l'utilisateur).

- Local (Uniquement le repo courant).

```
> git config --global user.name “John Doe”
> git config --global user.email name@email.com
> git config --global code.editor “code --wait”
// to edit the config file
> git config --global -e
// pour gérer le retour à la ligne et la fin de ligne. Sur Windows
'true', sur mac/linux 'input'.
> git config --global code.autocrlf true
```

On crée un dossier n'importe où sur sa machine et on se place dedans avec le terminal. Pour créer un nouveau repo vide :

```
> git init
```

Cela va créer un dossier caché «.git» qui contiendra l'arborescence du repo. On a pas besoin de toucher à ce dossier, il contient ce qui est géré automatiquement par git.

3. Workflow classique de Git

Rappel : Les stades de développement.

Quand on travaille sur un projet, on va modifier des fichiers dans un dossier. À certains moments on va vouloir enregistrer l'état actuel du projet, cela s'appelle créer un *commit*, la manipulation produit aussi un *snapshot*, c'est à dire une copie du projet au moment du commit.

Dans Git il y a une étape intermédiaire entre notre zone de travail et notre zone déployée. Cela s'appelle la zone de staging. Imaginez que vous travaillez sur un projet qui est déjà en ligne comme un thème Wordpress, vous allez vouloir vérifier que les changements fonctionnent dans cette zone de staging. Cela nous permettra aussi de choisir ce qu'on veut ajouter ou non au snapshot.

Pour ajouter des fichiers à la zone de *staging* :

```
> git add file1.php file2.css
```

On vérifie les fichiers en staging et on peut alors procéder au *snapshot* avec la commande suivante en ajoutant un commentaire. Ce commentaire est

important pour garder une historique compréhensive des changements dans notre code.

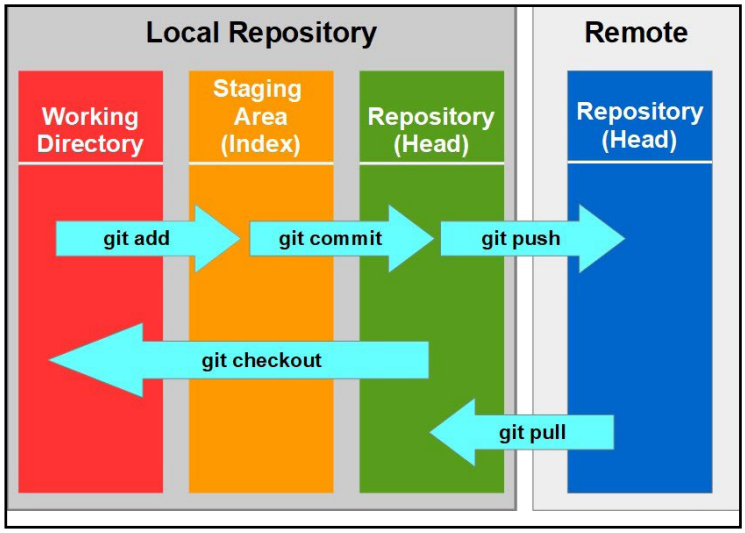
```
> git commit -m "Inital commit"
```

Pour autant, le staging n'est pas vide ! Quand on parle d'environnement de *staging*, **le staging contiendra toujours soit une copie conforme de ce qui est actuellement en production, soit de la prochaine version à déployer.** Cela veut dire que si on décide de supprimer un fichier côté *développement* (par exemple *file2.css*, on va devoir faire la commande *add file2.css* pour qu'il soit supprimé du *staging*.

```
> git add file2.css
À la ligne suivante, le -m veut dire message.
> git commit -m "Removed unused code"
```

Chaque *commit* contient des informations essentielles :

- Un ID généré automatiquement
- Le commentaire ajouté
- La date et l'heure du commit
- L'auteur du commit
- Un snapshot complet du projet au moment du commit.



Ce snapshot est fait de façon très efficace en terme de mémoire (compression, pas de duplication, etc.) En cas de catastrophe cela nous permet de rétablir une version fonctionnelle d'un programme.

Le diagramme ci-dessus ressemble à celui qu'on avait vu au chapitre dédié aux stades de développement mais ajoute les commandes de base de Git.

4. Exercice : Premier Commit

Exercice, tracking de tous les fichiers modifiés. On commence par créer quelques fichiers (par exemple index.html et style.css) dans notre dossier, on va ensuite vérifier le statut du repo.

```
> mkdir exercice
> cd exercice
> git init
> touch index.html style.css
> git status
```

Git va nous répondre que les changements ne sont pas trackés pour les fichiers que nous venons de créer, et qu'il n'y a pas encore eu de commits. On a plusieurs façons d'ajouter ces fichiers aux fichiers à tracker :

```
Pour ajouter juste les deux fichiers créés
> git add index.html style.css
Pour ajouter tous les fichiers html et css
> git add *.html *.css
// On peut ajouter le dossier entier avec un '.' mais ce n'est
probablement pas une bonne idée, on verra pourquoi plus loin.
> git add .
```

Pour cet exercice, nous allons tout de même utiliser cette dernière commande. Ensuite on va modifier notre fichier *index.html* et exécuter à nouveau la commande «status».

```
> echo "<html></html>" >> index.html
```

On voit que le fichier est marqué comme «modifié». Si on veut remplacer le fichier *index.html* du staging par celui du développement, on va à nouveau exécuter la commande «add», puis à nouveau «status» pour vérifier. Ensuite, on va «commit».

```
> git add index.html
> git status
> git commit -m "Ajout des balises de base html"
```

5. Bonnes pratiques du commit

On veut éviter de commit à chaque fois qu'on fait un changement, mais on veut également éviter de faire un énorme commit, on essaiera de trouver le juste milieu en arrivant à des étapes logiques du développement. La règle générale est donc de sauvegarder à des points logiques à la manière des checkpoints d'un jeu vidéo.

Il est parfois nécessaire de mettre un plus long message pour commenter notre commit. On peut alors utiliser la commande suivante pour laisser un plus long commentaire.

```
> git commit
```

Ce qui ouvrira l'éditeur configuré précédemment. Par convention, on laissera un résumé à la première ligne, suivi d'une description détaillée, avant le début des « # ».

```
Initial Commit

Ce commit corrige le bug xxx en modifiant la fonctionnalité
yyy...
#...
```

6. Add et commit en même temps

À ce stade, si vous avez compris la logique précédemment décrite, vous vous demandez peut-être pourquoi faire deux étapes différentes pour *add* et *commit*. Lorsqu'on travaille seul·e et qu'on progresse rapidement, ça peut effectivement être frustrant. C'est effectivement un workflow assez classique de Git. C'est aussi une source d'erreurs et c'est une bonne idée de garder cette commande pour quand on sera plus fluide avec Git.

```
> git commit -am "Un commit expéditif"
```

On voit dans la commande qui précède que pour add et commit en même temps, on va utiliser les flags -a et -m ensemble (donc -am).

7. Supprimer un fichier à plusieurs étages

Lorsqu'on veut supprimer un fichier du repo, il faut également le supprimer du staging. Pour supprimer ce fichier de notre environnement de développement,

on a même pas besoin de faire intervenir Git, on supprime simplement ce fichier de notre dossier. On l'a vu plutôt, on doit également supprimer ce fichier de notre staging en utilisant la commande add. Entre ces deux commandes on peut utiliser la commande git ls-files pour vérifier ce qui se trouve dans le staging.

```
> rm index.html
> git ls-files
> git add index.html
> git commit -m "Removed index.html"
```

Ce serait bien naturel de vouloir supprimer un fichier à la fois du stade dev et staging, c'est possible avec la commande suivante (qui équivaut à ce qui précède).

```
> git rm index.html
> git commit -m "Removed index.html"
```

8. Déplacer un fichier à plusieurs étages

De la même façon que la commande git rm, on peut déplacer/renommer un fichier à plusieurs étages. On utilisera à nouveau la commande classique du terminal mv mais au sein de git :

```
> git mv temp.html index.html
> git commit -m "New index page"
```

9. Ignorer des fichiers avec .gitignore

Ce qui donne sa raison d'être à la zone de staging c'est d'être un tampon entre nos environnements de travail et de production.

- Le code ne peut aller que vers le haut ;
- Les données ne peuvent aller que vers le bas.

Il est parfois nécessaire de travailler sur son code en utilisant les dernières données d'un site internet, mais il est primordial de ne pas utiliser Git pour ré-uploader des données ! Ceci peut mener à des situations catastrophiques, et c'est d'ailleurs régulièrement le cas, surtout sur des repos publics. Parmi les pires choses qui pourraient arriver :

- Uploader des identifiants/mots de passe ;
- Uploader des données de travail, des commentaires, des fichiers de clients ;
- Uploader des données sous licence propriétaire ;
- Remplacer de nouvelles données par de vieilles données périmées.

Afin de parer à cela, Git ne serait rien sans le fichier *.gitignore*.

10. Les branches

Un concept important de Git à l'intermédiaire entre le contrôle de version et la collaboration sont les *branches*, que l'on écrira à l'anglaise dans ce cours donc au singulier *branch*. Pour faire très simple, les branches nous permettent de faire des commits parallèles sans affecter le dépôt principal et puis de fusionner cette *branch* au dépôt principal lorsqu'une fonctionnalité est prête.

a. Master vs. Main

À l'origine, la branch principale d'un dépôt Git était appelée *Master*. Les différentes *Forges* (Github, Gitlab, SourceForge) peuvent toutefois choisir de changer le nom de cette branche. Depuis plusieurs années, Github utilise ainsi le nom *Main*, ceci pour les mêmes raisons que lorsque Adobe a remplacé les pages *Master* en page *Parent*, pour cesser d'utiliser des termes esclavagistes.

Le logiciel Git quant à lui, utilise toujours le terme *Master*. Ce qui posera problème lorsque nous utiliserons Git avec Github dans le futur.

Pour afficher le nom de la branche que l'on utilise actuellement, on utilisera la commande de base branch sans autre argument :

```
> git branch
```

Notons que d'autres noms de branche par défaut existe sur d'autres forges, comme *development*, *trunk*, etc.

b. Changer le nom de la branche par défaut

Comme tous les paramètres par défaut, le nom de la branche par défaut doit être réglé dans le fichier de configuration, au niveau global, avec la commande suivante :

```
> git config --global init.defaultBranch main
```

c. Renommer la branche principale d'un dépôt existant

Pour adapter notre dépôt existant, on aura besoin de le renommer. Par exemple, pour renommer notre branch master en main, on utilisera :

```
> git branch -m master main
```

d. Créer une nouvelle branch

Il existe plusieurs façons de créer une nouvelle branch. En utilisant la commande précédente on pourrait créer la nouvelle branch *darkmode* par exemple :

```
> git branch darkmode
```

Cette branche est créé mais nous sommes toujours en train d'éditer la branch main, pour changer la branch sur laquelle on se trouve, on utilisera :

```
> git switch darkmode
```

e. Fusionner deux branches

Pour fusionner deux branches, on commence par se placer dans la branch destinataire, habituellement *main*, puis on fusionne la branch avec le code à intégrer :

```
> git switch main
> git merge darkmode
```

Si la branch n'a plus d'utilité, on peut ensuite la supprimer :

```
> git branch -d
```

Dans cet exemple, la branche main n'a pas été modifiée. Git peut donc facilement comprendre qu'il n'y aura pas de conflit lors de la fusion, puisque le dernier commit de *main* est intégré dans la branche *darkmode*. Git fait alors un merge rapide, qu'on appelle un merge *Fast Forward* (Avance rapide).

On peut déjà supposer que si on avait modifié des choses dans la branche main

au même moment, on devra alors régler les conflits. De façon générale, si on exécute les commandes précédentes et que Git détecte ce genre de conflits, il nous permettra d'ouvrir un outil de résolution des conflits, dans notre cas au sein de notre éditeur *Visual Studio Code*.

f. Changer l'origine d'une branche divergente (rebase)

Maintenant que nous avons vu pratiquement tous les usages basiques du versioning avec Git, on va s'aventurer dans des cas plus particuliers qui nous seront de plus en plus utiles pour la suite du cours sur la collaboration.

Imaginons un autre cas de figure en ré-utilisant notre branche darkmode.

On crée notre branche darkmode et on travaille dans cette branche exclusivement à cette fonctionnalité. Pendant ce temps, un collègue ou nous-même continuons d'apporter des modifications au sein de l'application principale dans la branche main. On pourrait utiliser la commande merge depuis la branche darkmode, mais c'est une pratique problématique car cela va créer un commit dans un sens puis dans l'autre et l'historique de nos changements va devenir difficile à suivre.

On pourrait vouloir importer les nouveaux changements effectués dans main au sein de la branche darkmode sans effectuer un merge et un branch à nouveau.

Pour celà on va utiliser un autre type de fusion, au lieu de rapatrier la branche divergente dans notre branche de base, on va changer l'origine de la branche darkmode afin qu'elle se base sur un commit plus récent de base. C'est la commande rebase.

Lorsqu'on collabore ce sera d'autant plus utile puisqu'on veut s'assurer que le code avec lequel on contribue est basé sur une version récente du logicielle et pas sur une version datée.

Logiquement, on se place dans la branche secondaire et on exécute rebase vers la branche main :

```
> git switch darkmode
> git rebase main
```

IV. Synchroniser nos projets avec Github

1. Git en ligne et Github

Les dépôts Git peuvent être distribués, c'est à dire qu'ils peuvent exister à plusieurs endroits simultanément. Le fonctionnement en commits, avec une historique précise permet de fonctionner à plusieurs, de reprendre le code de quelqu'un d'autre, de contribuer, etc.

On peut utiliser ce fonctionnement sur sa propre machine ou sur un réseau interne d'entreprise. Il existe également des dépôts Git géants, hébergés sur internet, sur lesquels la plupart des développeurs, entreprises et projets publient leur code. Il en existait auparavant une multitude mais Github est rapidement devenu hégémonique. Github appartient aujourd'hui à Microsoft.

Une grande majorité du code open-source est accessible sur Github, qui offre une multitude d'autres services.

Github est utile aux développeur·euse·s pour une multitude de raisons :

- **Sauvegarder et synchroniser nos projets entre différentes machines.**
- **Avoir un portefolio** du code écrit, avec un aperçu aisé de la popularité, ou des technologies employées. Il permet également d'offrir un aperçu aux employeur·euse·s du nombre de projets sur lesquels un·e développeur·eus·e travaille, quels langages sont pratiqués, etc.
- **Veille technologique** : On peut aisément voir quels sont les projets, langages, logiciels à la mode, quelles technologies sont en train de percer, etc. Notamment via la page **Explore**. On peut également suivre les projets qui nous intéresse pour les voir apparaître dans notre fil d'actualité lorsqu'on s'y connecte, à la manière d'un réseau social.
- **Intégrer du code open-source** dans ces projets, trouver des technologies à utiliser soi-même dans ces projets, par exemple grâce aux pages **Awesome**.
- **Déployer gratuitement de petites applications** et de petits sites sans bases de données facilement. Par exemple via le service **Github Pages**.

Pour la suite du cours, vous aurez donc besoin de **créer un compte sur Github**.

2. Sauvegarder et synchroniser ses projets

Dans ce chapitre, nous allons introduire trois nouvelle commandes qui nous permettront dans un premier temps de collaborer avec nous-mêmes. C'est un workflow assez classique des développeurs : lorsqu'on utilise plusieurs machines (à l'école, au travail, chez soi,...) on voudra sauvegarder notre code

en ligne en terminant de travailler sur une machine et le restaurer en arrivant sur une autre. Ces trois nouvelles commandes sont :

```
git clone (Télécharger un repo distant)
git push (Pousser nos commits en ligne)
git pull (Télécharger nos commits localement)
```

a. Cloner un repo distant

Pour recopier n'importe quel dépôt Git sur une machine locale, on a même pas besoin d'un compte github, on utilisera simplement la commande clone. Clôner un repo distant est une commande très classique qui peut simplement servir à télécharger un dossier distant. De façon générale, on voudra également se placer dans le dossier que l'on vient de télécharger avec la commande linux adéquate, *cd*.

```
git clone https://github.com/user/repo.git
cd repo
```

b. Push & Pull

Push et Pull sont deux commandes essentielles lorsqu'on utilise Git en ligne. En anglais, ces mots signifient respectivement *Pousser* et *Tirer*, comme c'est écrit sur les portes de magasin.

La commande push va uploader tous les commits locaux qui ne sont pas encore dans notre repo Git en ligne.

De la même façon, la commande pull va télécharger les commits distants qui ne se trouvent pas sur notre machine locale.

c. En pratique, se connecter à notre premier repo

Dans cet exercice, on va :

- Créer un compte github si ce n'est déjà fait ;
- Créer un repo « First Repo » dans l'interface en ligne de Github ;
- Ce repo contiendra un fichier README.md et un fichier .gitignore ;
- On récupère l'URL type github.com/user/repo.git correspondante ;
- On ouvre VS Code en mode Remote WSL ;
- On crée un dossier /code dans notre répertoire linux, on se place à

l'intérieur et on lance la commande pour télécharger notre dépôt Git et on demande le statut pour vérifier que tout est OK :

```
git clone https://github.com/user/repo.git
cd repo
git status
```

- On crée un nouveau fichier index.html qu'on uploade sur Github :

```
touch index.html
git add .
git commit -m "Create index.html"
git push
```

Comme on peut le constater, lorsqu'on se trouve dans un dépôt Git à soi et qu'on ne change pas de *branch*, on peut éxécuter la commande *push* sans autres paramètres.

d. Tirer les changements

De la même façon, en arrivant sur une autre machine sur laquelle on travaille sur le même projet (déjà cloné), on se placera simplement dans le dossier du projet pour *tirer* les commits qu'on a poussé depuis la première machine.

```
git pull
```

e. Vérifier les changements

On pourrait facilement perdre le fil et se demander si on est synchro avec notre repo distant avant de faire un pull. Instinctivement, on pourrait tenter la commande status pour vérifier la différence.

```
git status
```

Mais de façon erronée, cette commande nous dirait « On branch main. Your branch is up to date with 'origin/main'. »

Ceci est du au fait qu'on a pas demandé à git d'aller vérifier le repo distant. Pour demander la liste des commits distants sans pour autant télécharger le contenu de ces commits, on pourrait exécuter la commande fetch, et puis la

commande status, qui nous donnera une toute autre réponse :

```
git fetch
git status
```

Status va alors nous retourner un message comme celui-ci : *Your branch is behind 'origin/main' by 7 commits, and can be fast-forwarded (use «git pull» to update your local branch)*

En fait, la commande pull est donc la contraction des deux commandes *fetch* et *merge*. De fait, Git vérifie si des commits existent, si oui, il les télécharge.

f. Conclusion

Une fois ces trois concepts compris, on sait maintenant comment collaborer avec soi-même pour éditer du code. L'étape suivante dans laquelle on collabore avec nos collègues parait couler de source. Il est important de bien maitriser ce workflow avant de passer à la suite.

3. Gestion des branches en remote

Lorsqu'on exécute les commandes clone, push et pull, on ne manipule pas les autres branches que la principale. D'ailleurs on peut tester ceci facilement en créant une branche sur github.com et en faisant la commande branch localement, celle-ci ne sera pas affichée par défaut. Il faudra ajouter le flag -a pour afficher les branches distantes.

```
git branch N'affiche que nos branches locales
git branch -a Affiche également les branches remote
git switch brancheDistante
```

On peut alors utiliser la commande switch normalement vers la branche distante qui sera téléchargée et connectée. On pourra donc faire les commandes push et pull vers celle-ci.

Dans le sens inverse, lorsqu'on crée une nouvelle branche localement, il faudra la pousser pour qu'elle soit synchronisée :

```
git push -u origin nouvelleBranche
```

Puisqu'on sait déjà comment fusionner une branche à une autre, il ne nous reste qu'à savoir comment supprimer une branche distante une fois que

la fonctionnalité a été fusionnée à notre branche main. Pour cela il faudra exécuter deux commandes.

```
git branch -d vieilleBranche
git push -d vieilleBranche
```

Il faudra évidemment avoir pris soin d'avoir fusionné tous les commits dans la branche main, locale et remote.

V. Divers

1. Rédiger des documentations avec Markdown

Vous connaissez déjà le HTML dont les initiales veulennt dire *HyperText Markup Language*, ou Language de Balisage Hypertexte. Une variante moins connue est le *Markdown*, un mot qui ne peut pas vraiment être traduit en français mais qui ressemble à une variante sans balises du HTML.

Les fichiers Markdown portent l'extension *.md*, contrairement aux fichiers HTML, ils ne comportent pas de Head. Les symboles utilisés en Markdown peuvent être maitrisés en quelques minutes et sont utiles pour la prise de note rapide puisqu'ils ne nécéssitent qu'un clavier (et pas de souris), et que les balises ne doivent pas être refermées.

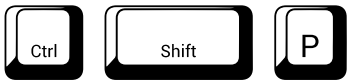
```
Titres jusqu'à 6 niveaux
# Titre H1
## Titre H2
### Titre H3

Mise en page classique
Un mot en **gras**
Un autre en *italique*
Des termes ***à la fois gras et italiques***

Listes non-ordonnées
- Les listes s'écrivent avec des tirets
- elles peuvent être indentées
        - Element
        - Element
Une liste ordonnée commence par des nombres mais ceux-ci n'ont pas
besoin d'être dans le bon ordre.
        1. Element
        2. Element
        1. Element
```

```
On peut créer des liens relatifs ou absolus :
Un [lien](https://google.com)
Et un [autre](contact.md)
```

Dans VS Code, on peut ouvrir la preview pour afficher immédiatement le Markdown, en utilisant la palette de commande :



.

De nombreux sites utilisent le Markdown, c'est le cas de Wikipédia et de Github. Ces sites proposent parfois une versions étendue de Markdown. Voir la version étendue de **la syntaxe markdown étendue pour Github ici.**

VI. Collaboration avec Github

Grâce au chapitre précédent, on connaît déjà toutes les bases pour travailler avec d'autres personnes sur du code avec Github. Pour ce qui concerne la collaboration, différentes organisations utilisent différents workflows. Ces workflows sont bien établis et documentés, il en existe de nombreux qui dépendent de nombreux critères comme la philosophie du projet, l'organisme qui le maintient, le nombre de contributeurs, etc.

1. Les différents workflows collaboratifs

a. Centralisé (SVN-style)

C'est le workflow « old-school » hérité des technologies qui ont précédé Git. Dans cette méthode de travail, on compte sur les fonctionnalités de Git pour équilibrer le chaos. Il n'y a qu'une seule branche, la *main*. Chaque dev utilise les commande clone et push jusqu'à ce que ça bloque. Si un·e dev tente de faire un push alors qu'un·e autre l'a déjà fait, iel devra faire un rebase sur le main.

C'est un workflow fonctionnel mais chaotique qui repose sur la puissance de Git. Les dev devront s'être entendu·e·s auparavant sur qui travaille sur quoi. Ce workflow n'est pas envisageable dans de grandes organisations.

Exemple :

- Sacha clone le repo puis travaille localement ;
- Camille clone le repo et travaille localement ;
- Sacha commit et push son travail sans problème ;
- Camille tente de push son code mais Git lui refuse ;
- Camille rebase sa branche depuis le main ;
- Camille push.

b. Feature Branch Workflow

C'est le workflow qui est décrit au chapitres sur les branches. Il est assez proche du modèle centralisé mais la séparation des tâches en branche clarifie les éventuels conflits.

c. Trunk Based Development (TBD)

Pour faire une explication très simplifiée, le TBD est une version moderne du Centralisé. Tout le monde développe sur la branche main. D'autres branches peuvent exister de façon éphémère. Les commits sont intégrés aussi rapidement que possible. Des tests et autres outils automatisés veillent à

éviter que le développement ne soit brisé.

C'est une pratique très utilisée dans les grosses entreprises parce qu'elle facilite la rapidité de déploiement.

d. Gitflow

Celle ci pourrait être définie comme une version avancée du Feature Branch Workflow. Une série de branches existent par défaut :

- **main** : Le code en production (déployé et utilisé par le public).
- **release** : Le code prêt à être mergé vers main.
- **develop** : Le code sur lequel on intervient, vers lequel on merge les features. Lorsqu'il est jugé production-ready, on le merge vers release.
- **hotfix** : Le code qui doit être intégré d'urgence à main en dépassant les autres branches.
- **feature** : Chaque feature a sa propre branche avant d'être intégrée à develop.

Ce workflow a l'avantage d'être prévisible et organisé et donc d'éviter les conflits et les bugs dans la version déployée d'un logiciel. Par contre, il est plus lent au sens où une fois qu'une fonctionnalié est réclamée ou un bug découvert il faut du temps pour qu'un correctif soit déployé.

Ce workflow a ses propres commandes (git flow ...) si l'on installe l'extension Git Flow (voir ici).