YouCode

Réaliser par

MELOUANE KHADIJA

Thème

« LES TROIS SCENARIOS DE GIT »

Encadré par :

Hanane Jabane

2019/2020

Table des matières :

I. Introduction : 2
II. Objectifs : 2
Premier scénario : 3
Etape 1 :Immertion 3
Etape 2 : découverte 4
Etape 3 : Historique 4
Etape 4 :Excluding files 5
Etape 5 :Branching and merging 6
Etape 6 : conflit résolution 7
Etape 7 : Merge Tools 7
Etape 8 : Challenge 8
Deuxième scénario : 8
Etape 1 : Tagging 8
Etape 2 : Stashing and Saving work in Progress : 8
Etape 3 : Voyage sur GitHub, Local Repo to GitHub Repo 9
Etape 4 : Mini challenge 9
Etape 5 : Création d'un local copy 10
Etape 6: Sending the website: 11
Etape 7 : Fetch and pull : 12
Troisième scénario : 13
Trello : 14
Conclusion : 15

I. Introduction :

Dans le cadre de la validation des compétences de la période SAS ; le brief projet est un moyen utile pour que vous validiez les compétences dans leur niveau respectif (N1, N2, N3)

La gestion des workflows sous GIT/GITHUB sera la base de notre apprentissage en terme de la gestion de projet agile.

Le brief projet est constitué d'un 3 scenarios diffèrent l'un de l'autre.

Un scenario doit être respecté en ce qui concerne le déroulement des taches sous un Product BACKLOG.

II. Objectifs :

- La formation d'un groupe de 5 apprenants
- Planification des taches selon une méthode de gestion de projet (SCRUM).
- La finalisation des Scénario dans les délais.
- La rédaction d'un rapport d'activités du groupe.
- La manipulation des commandes GIT.
- La découverte du monde GITHUB
- La gestion du temps, du stress et la finalisation des livrables.
- Les réunions de fin de Scénario
- La formalisation

Premier scénario :

Cette scénario est comporte 8 étapes, on utilise git bash en plus les commandes pour traiter les données de ce scenario.

Etape 1 :Immertion

Définition de Git :

Git est un Logiciel de contrôle de version qui permet de tracer l'évolution de projet et d'y apporter des modifications sereinement. Créé par Linus Thorvald. Il est utilisé pour gérer des codes sources.

Fonctionnement de fichier « .git » :

Quand on initialise le projet, un dossier caché « .git » est automatiquement créé. Ce dossier git contient toutes les informations nécessaires à votre projet dans le contrôle de version. Les informations sur les validations, et l'adresse du référentiel. Il contient également un journal qui stocke votre historique de validation afin que vous puissiez revenir à l'historique.
Tout d'abord on va créer un répertoire nommé « Projects », ensuit en va créer un repository local sous le nom "démo", puis en vérifier le status de répertoire démo par "git status".

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop
$ mkdir Projects
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop
$ cd Projects/
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects
$ mkdir Demo
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects
$ cd Demo/
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo
$ git init
Initialized empty Git repository in C:/Users/youcode/Desktop/Projects/Demo/.git/
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git status
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track
```

- Ce dernier commentaire signifie qu'il y a aucune modification au niveau de répertoire « Démo ».

Etape 2 : découverte

Dans cette phase, on découvre d'abord les fichiers de dossier « .git ».

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ cd .git
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo/.git (GIT_DIR!)
$ ls -al
total 13
```

Ainsi en définit quelques clauses de « .git » :

- HEAD : les informations sur la branche et les commit qu'il contienne etc.
- LOGS : l'historique des commits
- BRANCH : un moyen de demander un nouveau répertoire de travail, une nouvelle zone de stagging et un nouvel historique de projet.

Finalement nous avons créé dans la branche master un fichier nommé 'licence.md', puis nous avons commité ce fichier et afficher les fichiers traqués.

Etape 3 : Historique

Pour commencer cette étape, nous avons tout d'abord Affichez le dernier commit, toute en ajoutant l'option d'affichage de la hiérarchie de la branche, avec les commit et leur branche aussi. Nous avons fait ça avec la commande suivante:

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git log --all --graph --decorate
* commit 1284a5ac37ad1e969231711b5e43e56efb1d56b9 (HEAD -> master)
```

Ensuite nous avons utilisé l'alias pour remplacer le nom 'long' de cette commande.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ gir config -global historique "git log -all -graph -decorate
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
```

```
$ git config --list | grep alias
alias.historique=log --graph --oneline
alias.last=log -1 HEAD
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git historique README.md
* 7d9e493 Adding readme.md file
```

Etape 4 :Excluding files

A cette étape, il est demandé de renommer le fichier 'licence.md', et faire le staging avec mise à jour. Mais sans le faire avec la commande « git add . ». Par conséquence nous avons décidé de stagger avec « git add 'nom de fichier' » (il est possible aussi d'utilisé l'alias).

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ mv Licence.md Licence.txt
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ ls
Licence.txt  README.md
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git add Licence.txt
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git commit -m "Rename Licence.md to Licence.txt"
[master 55fdf8c] Rename Licence.md to Licence.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Licence.txt
```

Supposons que nous développons sur notre plateforme, surement on a des fichiers qu'on veut exclure de notre arborescence du repository Local.

Dans le repertoire locale on va créer un fichier nommé "appication.log" sans fait le stagging ,puis en créer un fichier nommé ".gitignore" et sur ce fichier en ajoute la ligne "*.loge" avec le stagging et le commit.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ touch application.log
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ touch .gitignore
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ code .
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git add .
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git commit -m "Creation de fichier application.log"
```

```
[master ebfd32e] Creation de fichier application.log
 2 files changed, 1 insertion(+)
commit ebfd32eb5ffc6ba8a69ca95f6b19b77dba410685 (HEAD -> master)
Author: majda FANNAN <fannanmajda123@gmail.com>
Date:   Tue Nov 26 15:24:55 2019 +0100
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git ls-files . --ignored --exclude-standard --others
application.log
```

- Après qu'on a fait le staging/commiting on a constaté que le fichier "application.log" n'est pas traqué.et c'est à cause de fichier. gitignore qui a pour rôle d'ignorer les fichiers que on veut pas visionner dans Git.

Etape 5 :Branching and merging

- The automatic : La fusion automatique est ce que le logiciel de contrôle de version fait lorsqu'il réconcilie les modifications survenues simultanément (dans un sens logique). De plus, d'autres logiciels déploient la fusion automatique s'ils permettent l'édition simultanée du même contenu.

- The manual : La fusion manuelle est ce à quoi les gens doivent recourir (éventuellement assistée par la fusion d'outils) lorsqu'ils doivent réconcilier des fichiers différents. Par exemple, si deux versions d'un fichier de configuration diffèrent légèrement d'un système à l'autre et qu'un utilisateur souhaite disposer des éléments les mieux adaptés, cela peut généralement être réalisé en fusionnant les fichiers de configuration à la main, en sélectionnant les modifications souhaitées à partir des deux sources (c'est également le cas. Appelé fusion bidirectionnelle). La fusion manuelle est également requise lorsque la fusion automatique est confrontée à un conflit de modifications.
- The fast forward merging :

C'est bien évidemment la phase merging. Nous avons fait des modifications sur le fichier « readme.md »au niveau de branche master et au niveau d'une nouvelle branche nommé 'updates'. Cependant nous ne nous avons pas commiter qu'après avoir basculé à la branche 'updates'

Constat :

*Au niveau de la branche :

On a constaté que la modification qu'on fait avant la création de la branche updates, se trouve dans cette dernière.

*Retournant à la branche master

On a constaté que aucune modification n'affecté la branche master.

Etape 6 : conflit résolution

Tout d'abord on a créé une branche qui s'appelle "BAD", Apres on a modifier le fichier README.md en ajoutant la ligne « trouble ». Et on a fait le stagging et le commiting on tapent la commande : « git commit –am «'message'»

Retournant vers la branche principale Master, maintenant on va faire une modification sur le même fichier README.md en ajoutant la ligne « troubleshooting ». Ensuite le staging et le commiting avec un message 'branche bien faite'.

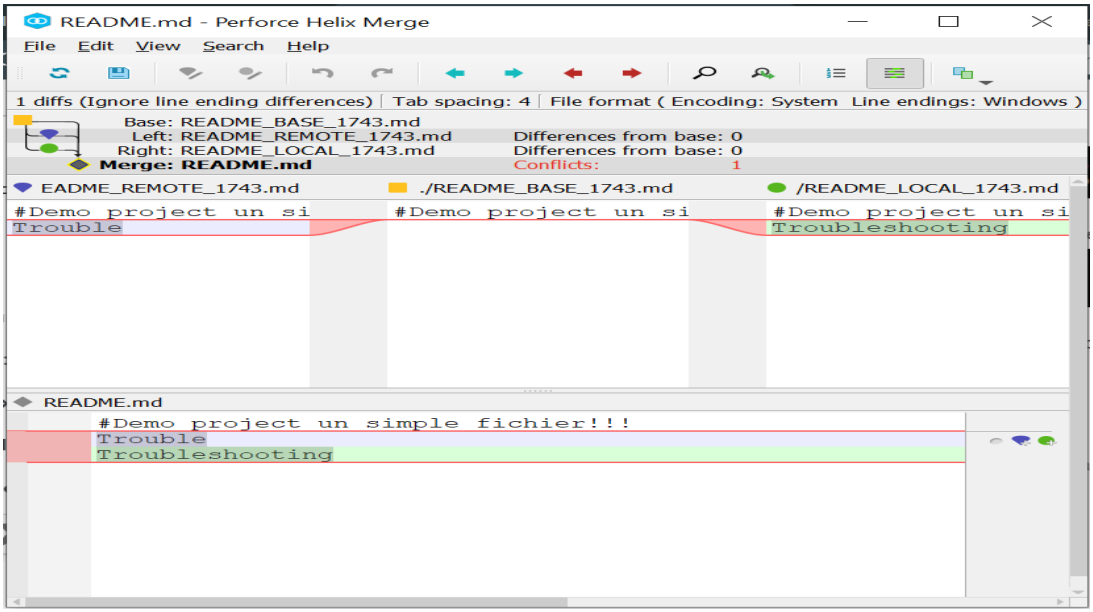
On utilise cette commande pour corriger le message de commit précédent

« git commit --amend -m "ajout de ligne trouble shooting" » On a constaté que le message de commit était changer . Après fusionner la branche ' BAD ' avec la branche principale 'Master ', on voit un message de conflit à cause de modification du même fichier README .md dans les deux branches.

Etape 7 : Merge Tools

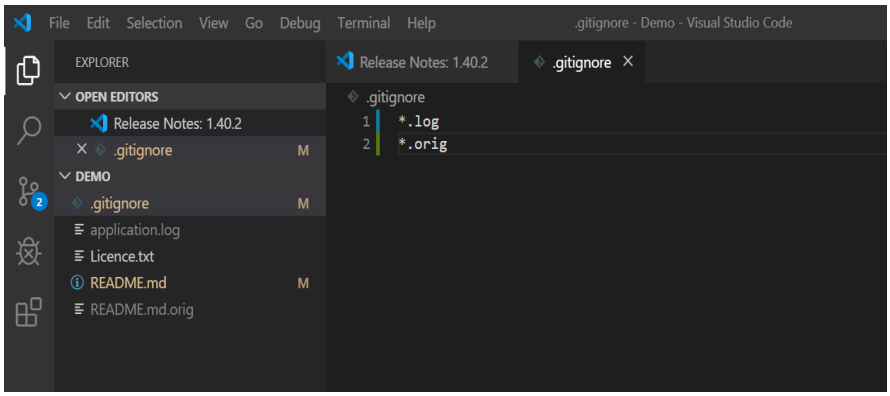
C'est le temps pour les mergetools. Nous avons fait le choix d'installer le programme p4merge outil de comparaison, celui-ci permet de repérer rapidement toutes les erreurs telles que les fautes de frappe, les espaces ou les lignes. Après l'installation, On a configuré git en ajoutant p4merge comme un mergetool. Ainsi on a édité la configuration de prompt au niveau de fichier. gitconfig.

On a constaté qu'après la configuration de prompt, si on tape la commande « git mergetool » L'interface de p4merge s'affiche. Une interface qui montre les conflits des deux branches.



Etape 8 : Challenge

Finalement, après avoir résolu les conflits entre les deux branches, il est convenable de rejeter les fichiers indésirables en laissant que les fichiers : readme.md /licence.txt et. Gitignore.



Deuxième scénario :

Ce scenario et comporte 7 étapes, on utilise le git bash et es commandes ainsi que GitHub.

Etape 1 : Tagging

Dans la branche principale "master" en va créer un TAG avec un nom V1.0 et un commentaire ‘ REALEASE 1.0 ‘, puis en fait afficher le TAG à travers la commande "git show".

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2
$ cd demo
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git checkout master
Already on 'master'
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git tag -a V1.0 -m "RELEASE 1.0"
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git show V1.0
tag V1.0
Tagger: Adam khairi <khairiadam2@gmail.com>
Date:   Thu Nov 28 16:22:27 2019 +0100

RELEASE 1.0
```

- Un TAG sert à crée une version de l’état actuel du projet.

Etape 2 : Stashing and Saving work in Progress :

Dans cette étape on a modifié le fichier README.md en ajoutant une ligne « Adding line », Apres on a utilisé Git stash cette commande nous permet de revenir au dernier commit sans perdre le brouillon de notre projet.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git stash
Saved working directory and index state WIP on master: 9877283 fin
```

A la suite on tape la commande git stash list qui nous donne la liste des stash et git status . Apres l'exécution de cette commande, on constate que les modifications de fichier README .md n'a pas enregistrer parce que on n'a pas commis avant le stash.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git stash list
stash@{0}: WIP on master: 9877283 fin scenario 1
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

On tape la commande git stash pop, On constate que cette commande permet de récupérer les données de stash.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working direct
        modified:   README.md
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (a285639bc85739bf298c12c2e2c84035ea496404
```

Etape 3 : Voyage sur GitHub, Local Repo to GitHub Repo

Commençons la nouvelle aventure de GitHub. L'intérêt de ce dernier est de d'offrir l'hébergement de projets avec Git, ainsi que la possibilité de suivre des personnes ou des projets.
Après que n'a ouvrit notre compte sur GitHub, on va créer un repo public et le cloner dans notre version locale

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git remote add origin https://github.com/adamkhairi/Projects-Gitub.git
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git remote -v
origin  https://github.com/adamkhairi/Projects-Github.git (fetch)
origin  https://github.com/adamkhairi/Projects-Github.git (push)
```

C'est l'heure de pousser le tout vers le serveur Github :

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ git push -u origin master --tags
```

pour pousser les tags aussi il ne suffit pas de faire le push de master mais aussi de ces derniers. On doit ajouter à la commande le nom du tag. En cas d'existence de plusieurs tags il faut simplement ajouter ' –tag'. Ainsi le –u « set-upstream » qu'on utilise dans la commande push a pour rôle d'ajouter une référence de suivi au serveur en amont sur lequel vous poussez

Etape 4 : Mini challenge

Après la création du dossier on va se déplacer dans ce dernier. Et maintenant on doit obtient le clé public et privé pour faire une communication sécure entre notre computer et github repo.

```
lenovo@DESKTOP-5KK8U9O MINGW64 ~/Desktop/project/demo/.SSH (master)
$ ssh-keygen -t rsa -C "elouadi.abdelali1997@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/lenovo/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/lenovo/.ssh/id_rsa.
Your public key has been saved in /c/Users/lenovo/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:4BZhaWeodxoY26B4dCZGb5VXxgwhpaWDgjrqGgNBz9E elouadi.abdelali1997@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
| o... ===*o      |
|..*.*E=*+.o      |
|o+.B+O*+         |
|o.oo+.++.        |
|+.   .o+S        |
|o.   ..          |
|+                |
|.o               |
|o.               |
+----[SHA256]-----+
```

D'après l'obtient les clés il faut donner notre GitHub repository la public clé comme dessus.
Et la dernière il faut change remote sur git bash de https à ssh la remote actuel est :

```
git remote -v
origin  https://github.com/abdelalilwafi/ssh.git (fetch
origin  https://github.com/abdelalilwafi/ssh.git (push
```

on change le remote par la commande suivante : git remote set-url
origin

Si on liste les remote on va trouver qu'elles sont changé

```
git remote -v
origin  git@github.com:abdelalilwafi/ssh.git (fetch)
origin  git@github.com:abdelalilwafi/ssh.git (push)
```

et dernièrement on va faire une communication entre github et le repo
pour examiner est ce que tous est bien on utilise git pull.

Etape 5 : Création d'un local copy

Sur GitHub créez un autre repo nommé (Monsiteweb)
Ajoutez à l'arboresence toujours sur github le fichier .gitignore et un fichier licence.txt ‘APACHE 2.0 ‘
Déplacez-vous dans le répertoire projets sous GIT

- Créez un clone github vers le local sous le nom (Monsiteweb-local)
- Vérifiez si le clone est créé.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/demo (master)
$ cd ..
Youcode@SpectRum MINGW64 ~/Desktop/Projects2
$ git clone https://github.com/adamkhairi/Monsiteweb1.git Monsitewb-local
Cloning into 'Monsiteweb-local'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0²), pack-reused 0
Unpacking objects: 100% (9/9), done.
Youcode@SpectRum MINGW64 ~/Desktop/Projects2
$ cd Monsiteweb-local/
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (maser)
$ ls -la
total 6
```

Etape 6: Sending the website:

Dans cette étape on va télécharger un site web boosttrap avec le fichier .htaccess et le fichier 404.html

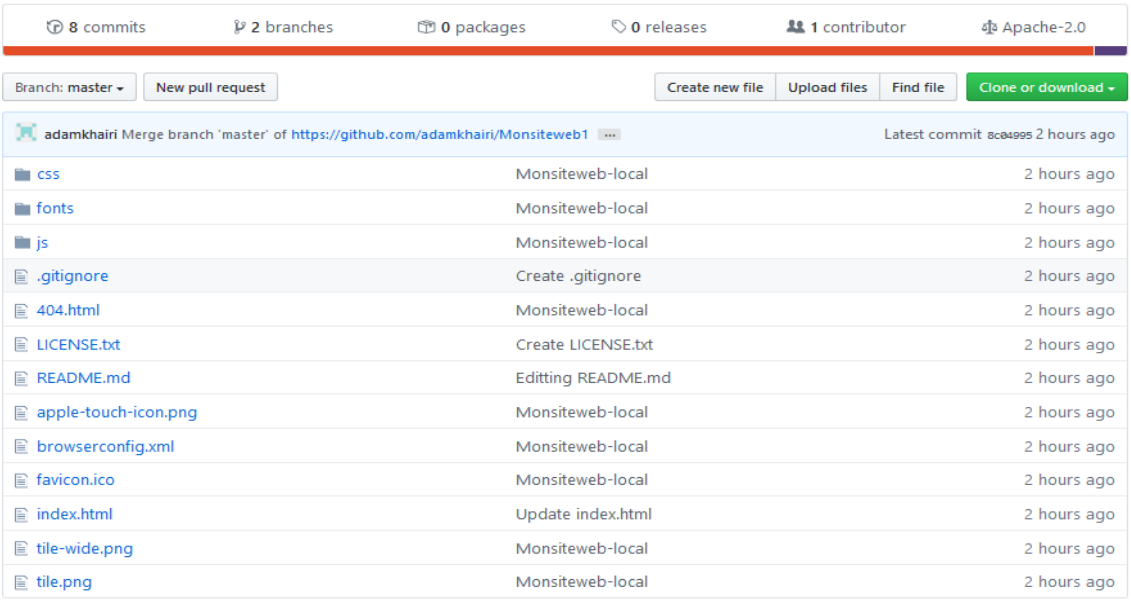
L'arborescence se compose des fichiers css, JavaScript, html, et des images qui construit le site web.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (maser)
$ cp -R ~/Desktop/initializr/* .
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (master)
$ ls -la
total 55
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:17 ./
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:12 ../
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:12 .git/
-rw-r--r-- 1 Youcode 197121     2 Nov 28 21:12 .gitignore
-rw-r--r-- 1 Youcode 197121  1272 Nov 28 21:17 404.html
-rw-r--r-- 1 Youcode 197121  3959 Nov 28 21:17 apple-touch-icon.png
-rw-r--r-- 1 Youcode 197121   416 Nov 28 21:17 browserconfig.xml
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:17 css/
-rw-r--r-- 1 Youcode 197121   766 Nov 28 21:17 favicon.ico
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:17 fonts/
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:17 img/
-rw-r--r-- 1 Youcode 197121  5283 Nov 28 21:17 index.html
drwxr-xr-x 1 Youcode 197121     0 Nov 28 21:17 js/
-rw-r--r-- 1 Youcode 197121 11558 Nov 28 21:12 LICENSE.txt
-rw-r--r-- 1 Youcode 197121    13 Nov 28 21:12 README.md
-rw-r--r-- 1 Youcode 197121  3482 Nov 28 21:17 tile.png
-rw-r--r-- 1 Youcode 197121  1854 Nov 28 21:17 tile-wide.png
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (master)
$ git status
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)
        404.html
        apple-touch-icon.png
        browserconfig.xml
        css/
        favicon.ico
        fonts/
        index.html
        js/
        tile-wide.png
        tile.png

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

Le stagging et le commit en une seule ligne « `git add . && git commit -m 'Monsiteweb-local'` » « `git commit –am 'message'` »

Push à GitHub on utilise « git push –u origin master »

Vérification du repo GitHub.



Etape 7 : Fetch and pull :

L'objectif de cette phase : c'est lorsqu'il y'a deux changements sur le repo local et sur le repo GitHub, donc les démarches décrites c'est pour la gestion de ce conflit.

Pour conclure ce scenario, nous allons découvrir de quoi s'agit-il le fetch et le pull. En premier, nous avons édité le fichier index.html sur GitHub et d'un autre côté nous avons modifié le fichier readme.md sur le repo local sans oublier bien sûr de commiter les deux fichiers.

Tapant maintenant la commande « git fetch » et ensuit « git statuts ».

- Constat :

Après avoir faire le fetch, toutes les commits effectués sur la branche courante et qui n'existaient pas encore dans la version locale ont été récupérés. Cependant ces données sont simplement stockées dans le répertoire sans être fusionnées.
En conséquence, on va faire le pull pour télécharger les données des commits de notre github et les fusionner avec notre repo local, et aussi le push pour envoyer les modifications locales apportées à notre la branche principale associée.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (master)
$ git pull
Merge made by the 'recursive' strategy.
 index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (master)
$ git push
```

On a terminé le scenario 2, Maintenant nous avons préparé l'environnement GITHUB pour des concepts plus avancés, ainsi que vous avez acquis comment dialoguer avec GITHUB en se basant sur les commandes GIT.

Troisième scénario :

Dans ce scénario on va voir le rôle de rebase et le merge et la différence entre eux :

Git Merge et Git Rebase ont le même objectif. Ils sont conçus pour intégrer les modifications de plusieurs branches en une seule. Bien que l'objectif final soit le même, ces deux méthodes y parviennent de différentes manières et il est utile de connaître la différence à mesure que vous devenez un meilleur développeur de logiciels.

Git Merge :

La fusion est une pratique courante pour les développeurs utilisant des systèmes de contrôle de version. Que des branches soient créés à des fins de test, de correction de bugs ou pour d'autres raisons, la fusion valide les modifications vers un autre emplacement. Pour être plus spécifique, la fusion prend le contenu d'une branche source et l'intègre à une branche cible. Dans ce processus, seule la branche cible est modifiée. L'historique de la branche source reste le même.

Avantages :

- Simple et familier
- Préserve l'histoire complète et l'ordre chronologique
- Maintient le contexte de la branche

Les inconvénients :

- L'historique des commits peut être pollué par de nombreux commits de fusion
- Le débogage à l'aide de git bisect peut devenir plus difficile

Fusionnez la branche principale dans la branche fonctionnelle à l'aide des commandes d'extraction et de fusion.

```
$ git checkout branch
$ git merge master
```

Git Rebase :

Rebase est un autre moyen d'intégrer les changements d'une branche à une autre. Rebase compresse tous les changements en un seul «patch». Ensuite, il intègre le patch sur la branche cible.

Contrairement à la fusion, le rebasement aplatit l'historique, car il transfère le travail terminé d'une branche à une autre. Dans le processus, l'historique non désiré est éliminé.

Avantages :

- Rationalise une histoire potentiellement complexe
- Manipuler un seul commit est facile (par exemple, les annuler)

- Evite les "bruits" de commit de fusion dans les dépôts avec des branches occupées
- Nettoie les validations intermédiaires en les transformant en une validation unique, ce qui peut être utile pour les équipes DevOps.

Les inconvénients :

- Réduire la fonctionnalité à une poignée de commits peut masquer le contexte
- Rebaser les référentiels publics peut être dangereux quand on travaille en équipe
- C'est plus de travail : utiliser rebase pour garder votre branche de fonctionnalités à jour en permanence
- Pour changer de base avec des branches distantes, vous devez forcer la poussée. Le plus gros problème auquel les gens sont confrontés est qu'ils forcent le push mais ne définissent pas le push par défaut. Cela se traduit par des mises à jour de toutes les branches portant le même nom, à la fois localement et à distance, ce qui est redoutable.

Rebasez la branche de fonctionnalité sur la branche principale à l'aide des commandes suivantes.

```
$ git checkout branch
$ git rebase master
```

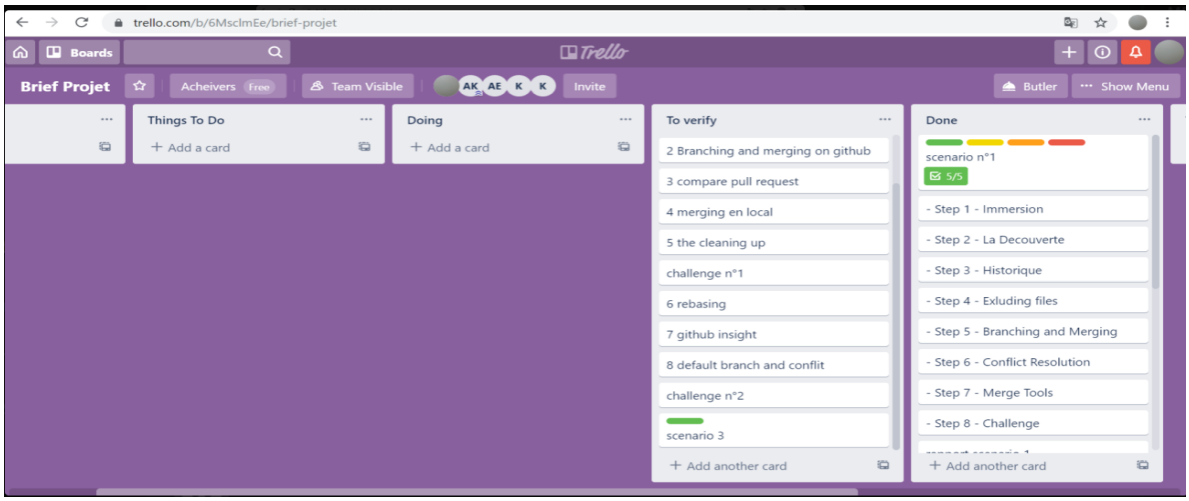
Rebase interactif

Le rebase interactif vous donne la possibilité de modifier des commits lorsqu'ils sont déplacés vers la nouvelle branche. Cette opération est plus efficace qu'un rebase automatisé, puisqu'elle permet de contrôler l'intégralité de l'historique des commits de la branche. Le plus souvent, le rebase interactif est utilisé pour nettoyer un historique désordonné avant de merger branche de fonctionnalité dans master.

Trello :

Trello est un outil de gestion de projet gratuit, simple et modulable, c'est-à-dire que nous pouvons nous organiser un peu comme nous le voulons, bien que ce logiciel soit inspiré de la méthode Kanban utilisé par Toyota.

Cet outil vous permet de vous organiser par tableau, par exemple, un tableau est égal à un projet ou un tableau est égal à un service de l'entreprise (commercial, comptabilité, marketing, etc…).



Scrum master :

Le Scrum Master anime une équipe de développement produit qui suit la méthode Scrum, une métaphore rugbystique nommée d'après la mêlée, dans laquelle une équipe s'organise elle-même et réagit rapidement aux changements.

product owner :

Est responsable de la définition et de la conception d'un produit. Il est chargé de mener à terme un projet en utilisant la méthode scrum*. Aussi appelé chef de projet digital, il est organisé et très rigoureux.

L'équipe scrum :

Une équipe Scrum comprend un chef de produit, une équipe de développement et un Scrum Master. Les équipes Scrum sont auto-organisées et pluridisciplinaires. Les équipes auto-disposées choisissent la meilleure façon d'accomplir leur travail, au lieu d'être dirigées par des personnes externes à l'équipe. Les équipes pluridisciplinaires ont toutes les compétences nécessaires pour effectuer le travail sans dépendre d'autres personnes n'appartenant pas à l'équipe.

Conclusion :

J'ai pris beaucoup de plaisir à en apprendre plus sur Git et à débloquer ses secrets. Il est très performant du fait d'avoir été écrit en C et optimisé pour le noyau Linux. La notion de Git Branching est terriblement efficace et permet de construire un projet de bout en bout de façon très simple et intuitive en minimisant les crises de conflits. Avec Git vous pouvez faire presque tout ce que vous voulez. La gestion des tags, branches, et merge est à la limite de la perfection en des temps records.

De plus, Github contribue largement à la notoriété de Git en fournissant un service d'hebergement de code open source permettant la visualisation du code, en ajoutant le côté social et relationnel.