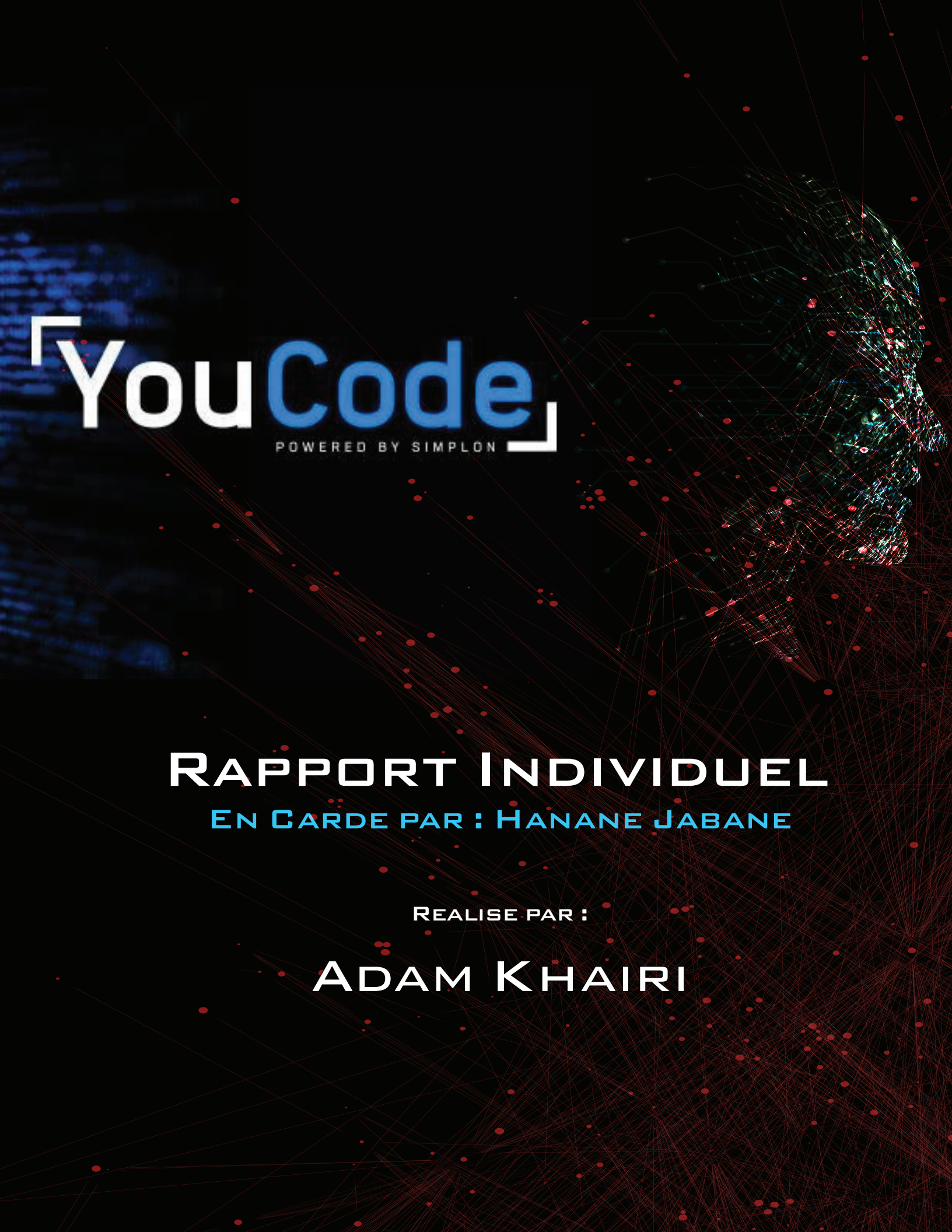


Rapport Individuel

En Carde par : Hanane Jabane

Realise par :

Adam Khairi

Sommaire

Introduction

Dans le cadre de la validation des compétences de la période SAS, Notre projet se déroule par 3 étapes :

Premièrement on a créé un tableau de « Trello » c'est un outil de gestion de projet en ligne. Et cet outil nous permet de suivre la progression des tâches de notre groupe et aussi de s'organiser facilement. Et l'utilisation de la méthode « SCRUM » est une méthode agile dédiée à la « gestion de projet ». Cette méthode de gestion, a pour objectif d'améliorer la productivité de notre équipe.

La deuxième étape est la réalisation du projet, nous évoquerons dans la partie conception le choix des commandes git et les résultats obtenus.

La troisième étape a été consacré pour la rédaction du rapport. Dans les ligne qui suivent nous allons détailler le développement de chacune de ces parties pour que le lecteur de ce rapport soit éclairé sur les différents étapes de ce brief projet.

Scenario 1

- 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.

Il est créé par Linus Torvalds.Il est utilisé pour gérer des codes sources.

```
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/
```

- Fonctionnement de dossier « .git » :

Quand on initialise un 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.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ cd .git

youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo/.git (GIT_DIR!)
$ ls -al
total 13
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:40 ./
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:37 ../
-rw-r--r-- 1 youcode 197121  22 nov.  26 11:40 COMMIT_EDITMSG
-rw-r--r-- 1 youcode 197121 130 nov.  26 11:36 config
-rw-r--r-- 1 youcode 197121  73 nov.  26 11:36 description
-rw-r--r-- 1 youcode 197121  23 nov.  26 11:36 HEAD
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:36 hooks/
-rw-r--r-- 1 youcode 197121 137 nov.  26 11:40 index
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:36 info/
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:40 logs/
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:40 objects/
drwxr-xr-x 1 youcode 197121   0 nov.  26 11:36 refs/
```

- Add et commit des Changements :

Après la création d'un fichier il faut le transférer au Staging Area et le committer (créer un commentaire) pour enregistrer les modifications par les commandes suivantes :

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git add .

youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git commit -m "Adding readme.md file"
[master (root-commit) 7d9e493] Adding readme.md file
```

- Excluding files :

Il est possible d'exclure les fichiers pour ne pas passer à la zone de Staging, cela par créer un fichier nommé «. gitignore » et de mettre l'extension du fichier indésirable .

- Branching and merging :

Les branches permettent de travailler sur des versions qui divergent de la branche principale contenant votre projet courant, en peut créer une Branch et basculé vers cette dernière en tapent les commandes suivantes :

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git branch updates

youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git checkout updates
Switched to branch 'updates'
M       README.md
```

Lorsque on travaille sur plusieurs branches fusionner des branches est une pratique courante lorsque vous travaillez sur un projet : vous devez toujours chercher à remettre les modifications faites sur vos différentes branches dans la branche principale **master,** et pour le faire on utilise la commande :

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git merge updates
Updating ebfd32e..971cc3a
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Il arrive très souvent qu'il y ait des conflits entre les deux branches qui empêchent de les fusionner, par exemple lorsque plusieurs personnes travaillent en même temps sur un même fichier.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master)
$ git merge BAD
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Après le fusionnement on voir que le message nous donne un conflit, Apres l'exécution de la commande cat README.md le contenu du fichier readme.md affiche les deux modification qu'on a déjà fait .pas conséquence un conflit s'est engendrés.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master|MERGING)
$ cat README.md
#Demo project un simple fichier (modifications1)
<<<<<<< HEAD
troubleshooting
=======
trouble
>>>>>>> BAD
```

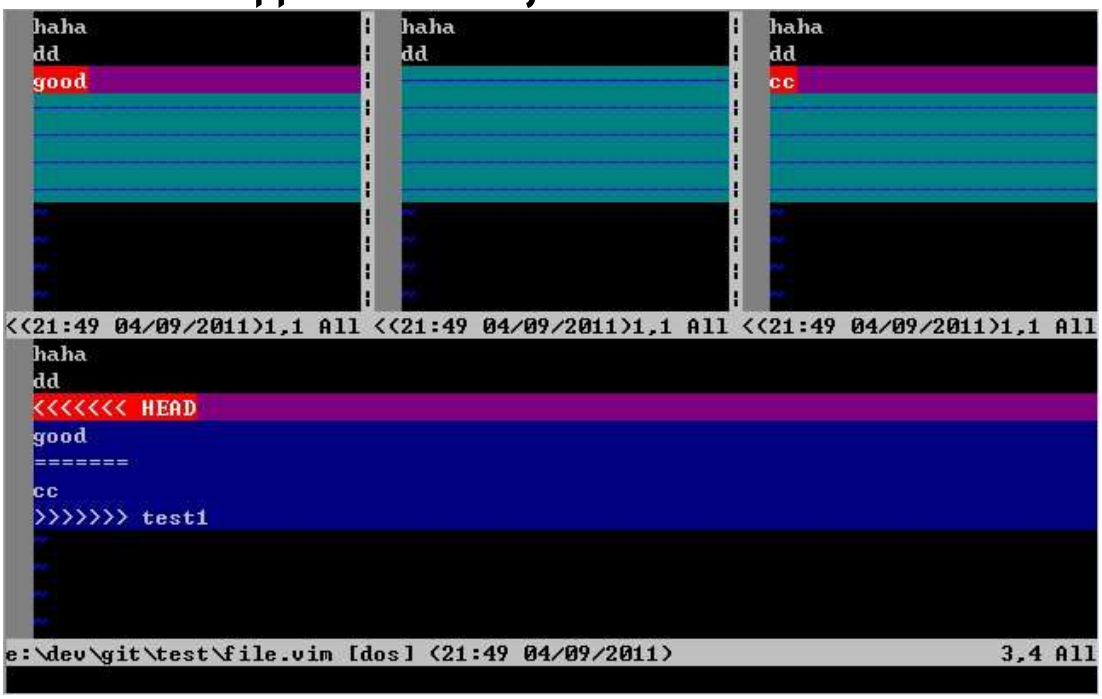
- ## Mergetool vimdiff

Ici nous avons résolu le conflit en ouvrant directement le fichier posant problème dans la console. Sachez qu'il existe aussi des outils proposant des interfaces graphiques pour comparer les différences de versions d'un fichier.

```
youcode@DESKTOP-OC51QH5 MINGW64 ~/Desktop/Projects/Demo (master|MERGING)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge
araxis bc codecompare smerge emerge vimdiff
Merging:
README.md

Normal merge conflict for 'README.md':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):
```


haha
dd
good
haha
dd
haha
dd
cc
<<(21:49 04/09/2011)1,1 All <<(21:49 04/09/2011)1,1 All <<(21:49 04/09/2011)1,1 All
haha
dd
<<<<<<< HEAD
good
=======
cc
>>>>>>> test1
e:\dev\git\test\file.vim [dos] (21:49 04/09/2011) 3,4 All

Scenario 2

- Tagging :

Git donne la possibilité d'étiqueter un certain état dans l'historique comme important. Généralement, les gens utilisent cette fonctionnalité pour marquer les états de publication (v1.0 et ainsi de suite).

Pour lister les étiquettes existantes dans Git par `git tag` , et pour créer un Tag en utilise la commande suivante :

```
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
```

Afficher un historique clair des derniers commits :

```
$ git config --global alias.history 'log --pretty=format:"%h %ai | %s%d [%an
```

- Fetch and Pull :

Les deux commandes permettent de mettre à jour un répertoire local avec les données d'un repository distant. Elles n'ont cependant pas le même fonctionnement.

La commande **git fetch** va récupérer toutes les données des commits effectués sur la branche courante mais ne seront pas fusionnées avec votre branche locale.

La commande **git pull** est en fait la commande qui regroupe les commandes git fetch suivie de git merge, Cette commande télécharge les données des commits qui n'ont pas encore été récupérées dans votre branche locale puis fusionne ensuite ces données.

- Alias :

le principe de l'alias est de permettre d'exécuter une (ou des) commande(s) longue(s) sans avoir à taper la commande complète, mais plutôt un nom raccourci, et pour créer un Alias qui afficher un historique clair des derniers commits :

```
$ git config --global alias.history 'log --pretty=format:"%h %ai | %s%d [%an
```

Scenario 3

- **Rebase :**

Comme son nom l'indique rebase permet de déplacer une branche et de changer son commit de départ, le principe est très simple, on déplace nos commits. En revanche git va réécrire l'historique, supprimer nos anciens commits et recréer nos nouveau commits à partir de la nouvelle base.

```
Youcode@SpectRum MINGW64 ~/Desktop/Projects2/Monsiteweb-local (master)
$ git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: changing title
```

Conclusion

J'ai pris beaucoup de plaisir à en apprendre plus sur Git et à débloquer ses secrets. 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.