Partie 4 : Mettre en œuvre des outils de gestion de versions et de mesure de la qualité du code

Filière : Développement digital – option web full stack

Module : Approche agile

PARTIE 4

Mettre en œuvre des outils de gestion de versions et de mesure de la qualité du code

Dans ce module, vous allez :

- Manipuler les outils de gestion de versions (Git/Gitlab) .
- Manipuler l'outil de mesure de la qualité du code (SonarQube)

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

Ce que vous allez apprendre dans ce chapitre :

- Présentation des outils existants de gestion
- Présentation de Git et Gitlab et comparaison entre les deux
- Présentation des fonctionnalités de Gitlab
- Installation et manipulation de Git.
- Manipulation des commandes de base de Git (Git bash)
- Notion de branches et gestion des conflits de fusion avec Git

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. **Intérêt de la gestion de version et présentation des outils existants de gestion de versions**
2. Présentation de Git
3. Présentation de Gitlab
4. Manipulation des dépôts avec Gitlab
5. Gestion des conflits de fusion avec Git/GitLab
6. Comparaison Git vs Gitlab

01-Manipuler les outils de gestion de versions (Git/Gitlab) :

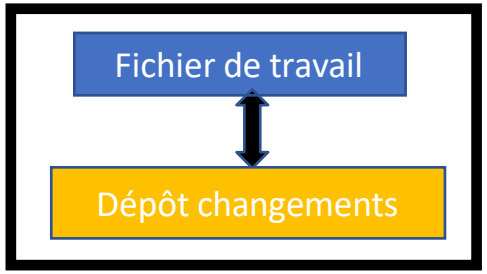
Gestion de versions

PARTIE 4

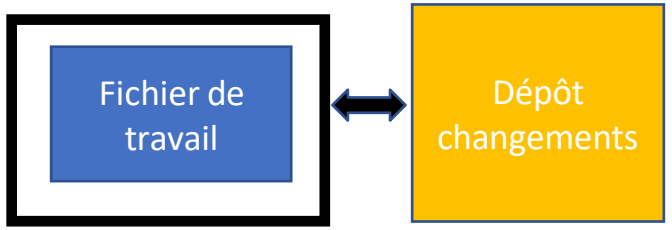
C'est quoi la gestion de versions?

- La gestion de versions également appelé historique de versions ou contrôle de versions (en anglais *version control system* (VCS)) se réfère, dans le milieu numérique, au stockage de plusieurs versions de fichier(s) afin de pouvoir tracer l'évolution chronologique à travers les changements apportés d'une version à l'autre.
- Elle peut s'appliquer à un fichier individuel (exemple : un document texte) ou à plusieurs fichiers d'un projet; elle peut se faire au niveau individuel ou dans des groupes
- La gestion de versions se fait aussi à la base de plateformes collaboratives privées ou open source, comme par exemple GitHub ou GitLab.

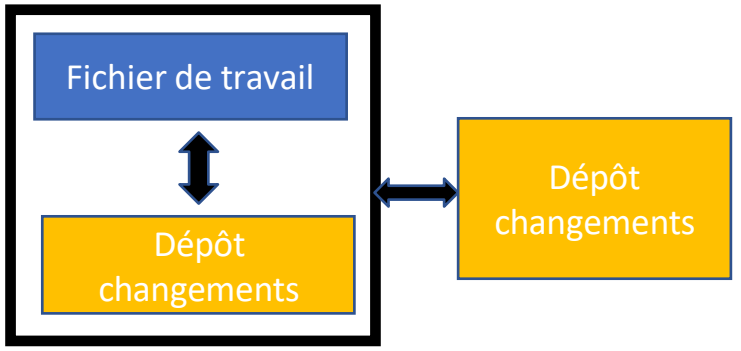
- On peut diviser les systèmes de gestion de versions en trois catégories :
  - Gestion de versions locale,
  - Gestion de versions centralisée,
  - Gestion de versions distribuée ou décentralisée.



Dans un système de gestion de versions **locale**, le *dépôt(code ou code de source)* avec les changements se trouve physiquement sur la même machine qui stocke les fichiers tracés



Dans la gestion de versions **centralisée**, le *dépôt* qui contient les informations sur les changements se trouve sur une autre machine par rapport aux fichiers de travail. Le cas de figure le plus commun consiste à garder le *dépôt* des changements sur un serveur centralisé, tandis que les différents ordinateurs individuels des personnes qui participent au projet ne gardent que la dernière version des fichiers de travail.



La gestion de versions **distribuée** combine la gestion de version locale et centralisée en créant ***deux*** *dépôts* des changements :

1. Le premier se trouve sur la même machine des fichiers de travail
2. Le deuxième se trouve dans une autre machine, souvent un serveur ou une plateforme *cloud* comme GitHub ou GitLab, qui s'occupe de centraliser les changements

gestion de versions distribuées

- La gestion distribuée est notamment l'un des éléments principaux des projets open source, comme elle est représentée dans l'image ci-bas
- La gestion de versions décentralisée consiste à voir l'outil de gestion de versions comme un outil permettant à chacun de travailler à son rythme, de façon désynchronisée des autres, puis d'offrir un moyen à ces développeurs de s'échanger leur travaux respectifs. De ce fait, il existe plusieurs dépôts pour un même logiciel.
- Ce système est très utilisé par les logiciels libres.
- Par exemple, **GNU Arch** et **Git** sont des logiciels de gestion de versions décentralisée.

Exemple de gestion de versions distribuée source https://edutechwiki.unige.ch/

- Dans ce schéma qui utilise **Git** et **GitHub** comme exemple (le même principe peut-être appliqué à d'autres systèmes), le dépôt local se trouve dans un dossier **.git** sur l'ordinateur d'une personne. Ce dépôt local est partagé en open source à travers un dépôt central qui se trouve sur la plateforme *cloud* GitHub. Ce dépôt central peut être utilisé par plusieurs personnes pour :
  - Contribuer au même projet: envoyer des changements qui sont ensuite incorporés dans le dépôt central et peuvent donc être ensuite propagés à tout dépôt local connecté,
  - Utiliser ce dépôt central comme base pour un autre projet.

Exemples d'outils de gestion de versions

- **Wiki** : appelée dans ce contexte souvent historique des versions, est l'une des fonctionnalités principales caractérisant tout système qui s'inspire du principe wiki. Des systèmes de ce type peuvent se trouver à plusieurs endroits, avec des fonctionnalités légèrement différentes, comme par exemple dans :
  - **Moodle** : une plateforme pédagogique très utilisée dans la formation et l'enseignement,
  - **GitHub** : dans lequel le wiki est souvent utilisé comme documentation ou moyen d'organisation entre collaborateurs,
  - **Tous les sites qui sont basés sur un moteur wiki**, comme par exemple MediaWiki, le logiciel open-source à la base de Wikipédia et également de EduTech Wiki.
- **Logiciels de traitement de texte :** plusieurs logiciels de traitement de texte, surtout dans les versions en ligne et collaboratives, permettent de retracer l'historique des versions : OFFICE ,GOOGLE DOCS..
- **CVS (Concurrent Versioning System)** : fonctionne sur un principe centralisé, de même que son successeur SVN (Subversion).
- **Logiciels de SCM décentralisés** : sont apparus plus récemment : Mercurial et surtout Git, que nous utiliserons dans la suite de ce chapitre. Ce sont également des logiciels libres.
- **TFS (Team Foundation Server)** de Microsft: TFS est une solution payante qui fonctionne de manière centralisée.

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. Intérêt de la gestion de version et présentation des outils existants de gestion de versions
2. **Présentation de Git**
3. Présentation de Gitlab
4. Manipulation des dépôts avec Gitlab
5. Gestion des conflits de fusion avec Git/GitLab
6. Comparaison Git vs Gitlab

01- Manipuler les outils de gestion de versions(Git/Gitlab) :

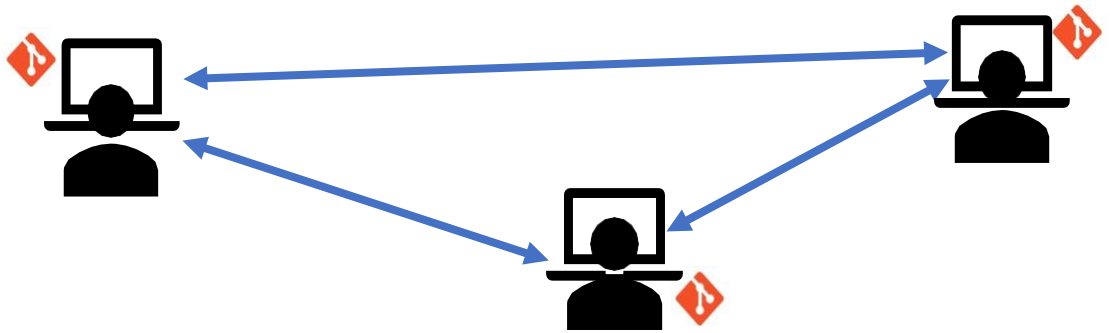
Présentation de git

C’est quoi git?

- **Git** est un logiciel libre de gestion de versions. C'est un outil qui permet d'archiver et de maintenir les différentes versions d'un ensemble de fichiers constituant souvent le code source d'un projet logiciel, il est multi-langages et multi-plateformes. **Git** est devenu à l'heure actuelle un quasi-standard .

logo de Git (Le nom "Git" se prononce comme dans guitare)

- **Git** rassemble dans un **dépôt (*repository* ou *repo*)** l'ensemble des données associées au projet. Il fonctionne de manière décentralisée: tout dépôt Git contient l’intégralité des données (code source, historique, versions, etc).
- Chaque participant au projet travaille à son rythme sur son dépôt local. Il existe donc autant de dépôts que de participants.
- **Git** offre des mécanismes permettant de synchroniser les modifications entre tous les dépôts.



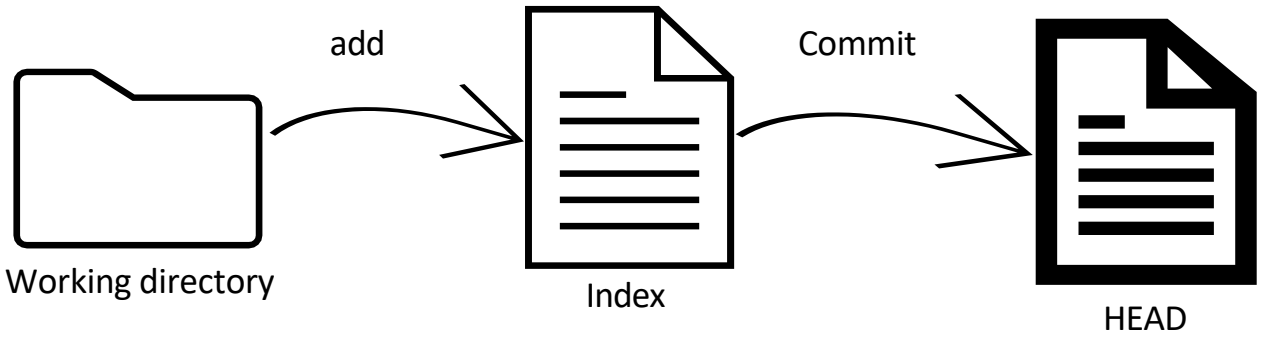
PARTIE 4

01- Manipuler les outils de gestion de versions(Git/Gitlab) :

Présentation de git

Les dépôts avec git:

- **Version :** contenu du projet à un moment de son cycle de vie.
- **Dépôt**(repository) : l'historique du projet, contenant toutes ses versions.
- **Branche** (branch) = variante d'un projet.
- Un dépôt Git correspond physiquement à un ensemble de fichiers rassemblés dans un répertoire .**git**. Sauf cas particulier, il n'est pas nécessaire d'intervenir manuellement dans ce répertoire.
- Lorsqu'on travaille avec Git, il est essentiel de faire la distinction entre trois zones :
    - Le **répertoire de travail** (*working directory*) :correspond aux fichiers actuellement sauvegardés localement.
    - L'**index** ou *staging : area* est un espace de transit.
    - **HEAD :** correspond aux derniers fichiers ajoutés au dépôt.



PARTIE 4

01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Installation de git

Installation git?

1. Vérifier dans le terminal que git est bien installé: **git version**.

```
C:\Users\[redacted]>git version
git version 2.[redacted].1.windows.1
```

2. Si ce n'est pas le cas, installez-le en suivant les recommandations de votre système d'exploitation où en le téléchargeant depuis git-scm.com/downloads, puis redémarrez votre terminal.

Git Bash

Lancer git après l'installation et tapez la commande **git** pour voir les différentes commandes git et leurs rôles

```
DELL@DESKTOP-01DOJ3U MINGW64 ~/Desktop
$ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone     Clone a repository into a new directory
   init      Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add       Add file contents to the index
   mv        Move or rename a file, a directory, or a symlink
   restore   Restore working tree files
   rm        Remove files from the working tree and from the index
```

PARTIE 4

Principales commandes git

Les principales commandes git sont :

- **git init** : crée un nouveau dépôt vide à l'emplacement courant.
- **git status** : affiche les différences entre le répertoire de travail, l'index et HEAD.
- **git add** : ajoute des fichiers depuis le répertoire de travail vers l'index.
- **git commit** : ajoute des fichiers depuis l'index vers HEAD.
- **git log** : affiche l'historique des commits.
- **git clone** : clone un dépôt existant local ou distant.
- **git pull** : récupère des modifications depuis un dépôt distant vers HEAD.
- **git push** : publie des modifications depuis HEAD vers un dépôt distant.

PARTIE 4

Exemple des principales étapes pour utiliser les commandes git localement

1. Ouvrez git bash ;
2. Déplacez vous vers un dossier de votre choix à l'aide de la commande **cd: ex. cd E:/**
3. Créez un nouveau dossier : **mkdir mon_projet**
4. Placez vous dans ce dossier: **cd mon_projet**
5. Initialisez le dossier avec : **git init**
6. Créer un fichier texte d'exemple: **echo « Bonjour tout le monde » > file.txt**
7. Vous pouvez proposer un changement (l'ajouter à l'**Index**) en exécutant les commandes **git add <fichier>** ou **git add * .**
8. Pour valider ces changements, utilisez **git commit –m 'Message de validation'** .
9. Maintenant, vous pouvez envoyer vos changements vers le serveur distant : en utilisant les commandes 7 et 8 à chaque changement .
10. Pour afficher l'historique des commit, utilisez **git log** ou **git log --oneline**

PARTIE 4

Branches

Une branche dans Git est simplement un pointeur léger et déplaçable vers un de ces commits. La branche par défaut dans Git s'appelle master. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. À chaque validation, le pointeur de la branche master avance automatiquement.

Les branches sont utilisées pour développer des fonctionnalités isolées des autres. La branche *master* est la branche par défaut quand vous créez un dépôt. Utilisez les autres branches pour le développement et fusionnez ensuite à la branche principale quand vous avez fini.

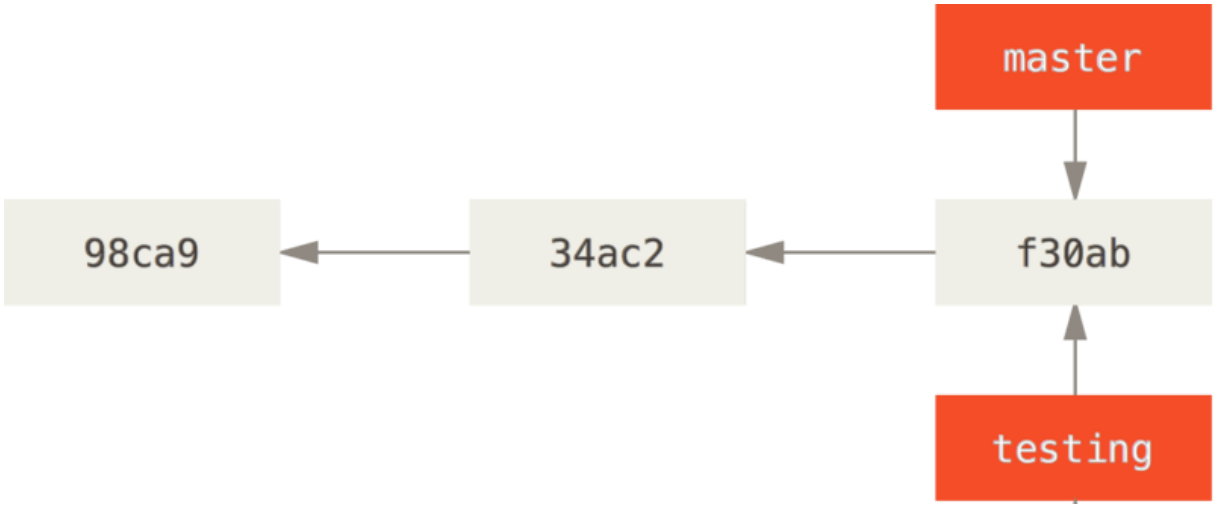
PARTIE 4

Créer une nouvelle branche

Que se passe-t-il si vous créez une nouvelle branche ? Eh bien, cela crée un nouveau pointeur pour vous.
Supposons que vous créez une nouvelle branche nommée **`testing`**. Vous utilisez pour cela la commande `git branch` :

```
$ git branch testing
```

Cela crée un nouveau pointeur vers le **`commit`** courant.



Deux branches pointant vers la même série de *commits*

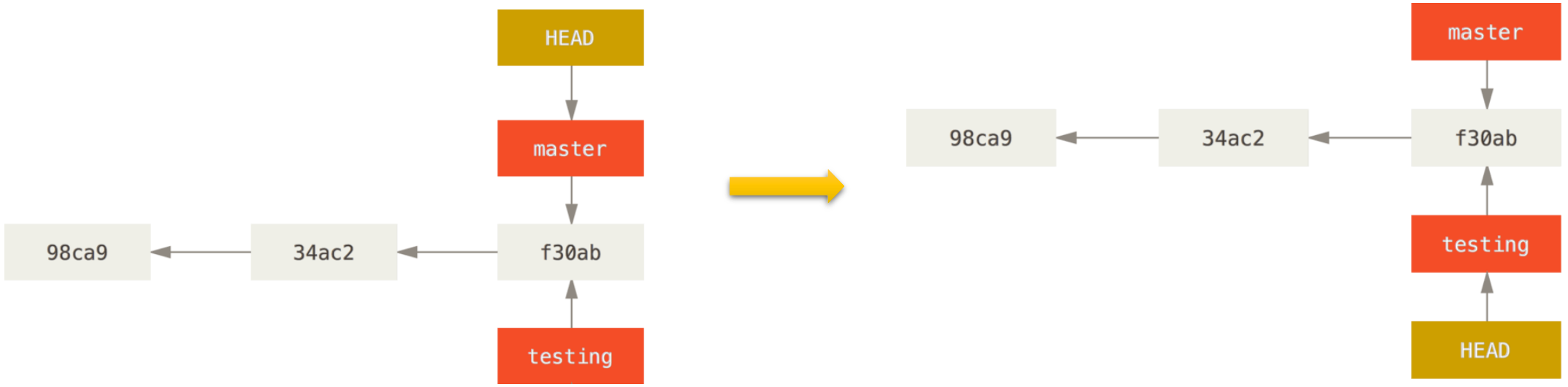
PARTIE 4

Basculer entre les branches

- Comment Git connaît-il alors la branche sur laquelle vous vous trouvez ? Il conserve à cet effet un pointeur spécial appelé **HEAD**. Dans Git, il s'agit simplement d'un pointeur sur la branche locale où vous vous trouvez.
- Dans ce cas, vous vous trouvez toujours sur master. En effet, la commande *git branch* n'a fait que créer une nouvelle branche — elle n'a pas fait basculer la copie de travail vers cette branche.
- Pour basculer sur une branche existante, il suffit de lancer la commande *git checkout.*
- Basculons sur la nouvelle branche testing :

$ git checkout testing ou **$ git switch testing**

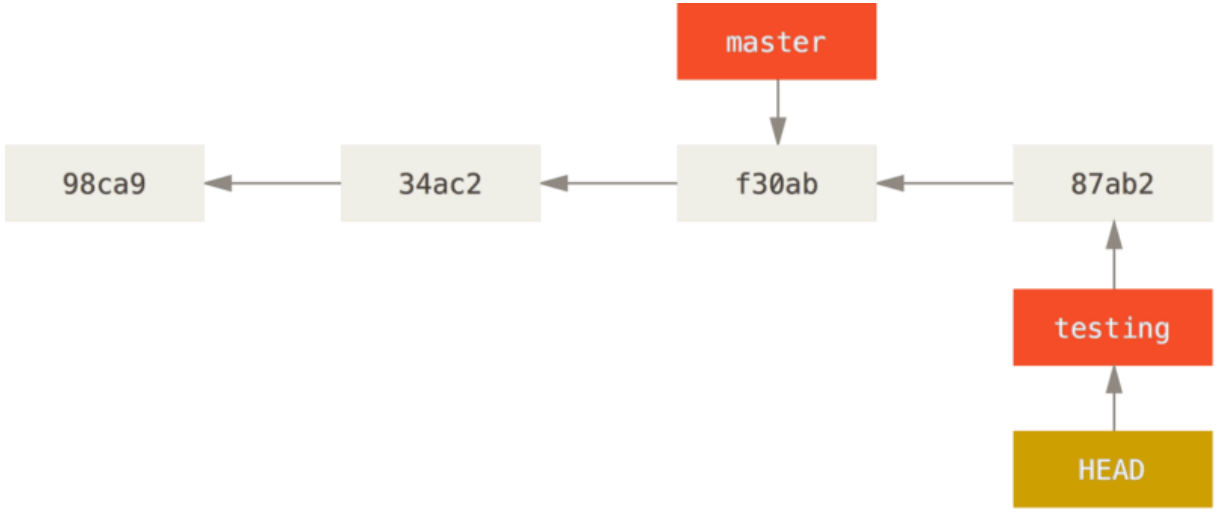
Cela déplace HEAD pour le faire pointer vers la branche testing.



Basculer entre les branches

```
$ vi test.rb

$ git commit -a -m 'made a change’
```

Si on effectue un changement et on fait un commit, La branche `testing` avance tandis que la branche `master` pointe toujours sur le **`commit`** sur lequel vous étiez lorsque vous avez lancé la commande `git checkout` pour changer de branche.



PARTIE 4

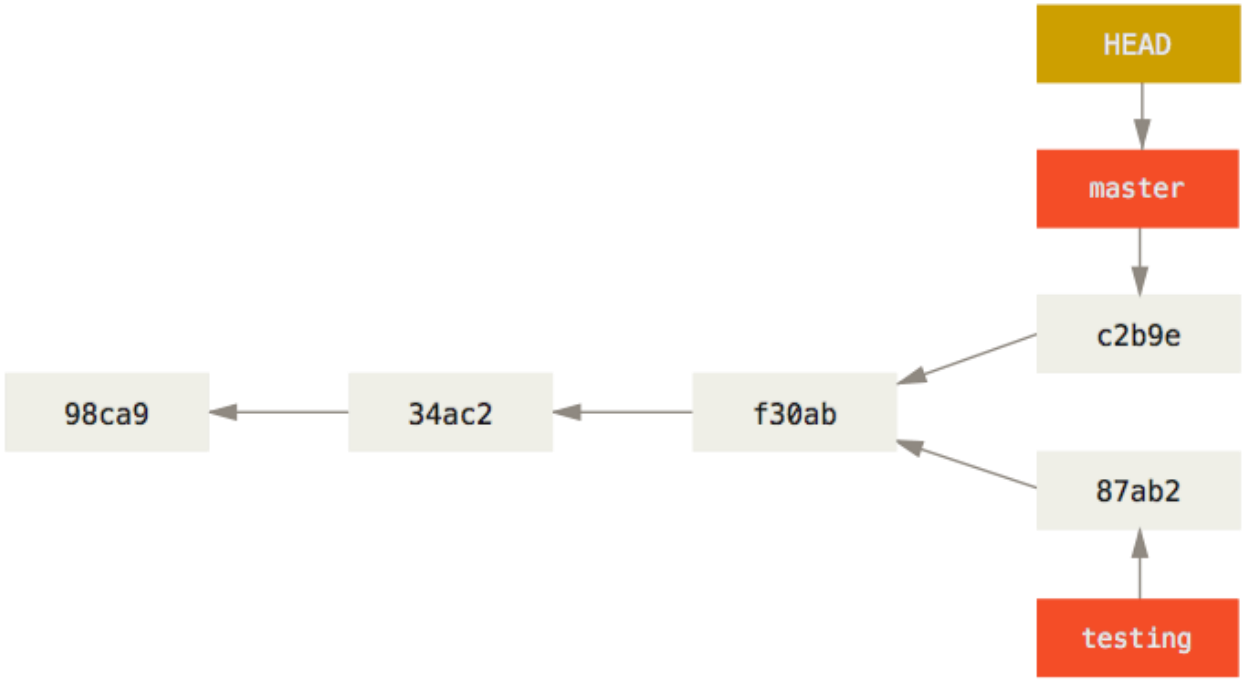
PARTIE 4

Basculer entre les branches

Rebasculant vers la branche master et effectuant des changements:

```
$ git checkout matser
$ vi test.rb
$ git commit -a -m 'made other changes'
```

Ces deux modifications sont isolées dans des branches séparées : vous pouvez basculer d'une branche à l'autre et les fusionner quand vous êtes prêt.



PARTIE 4

Créer une branche et basculer dessus

Il est habituel de créer une nouvelle branche et de vouloir basculer sur cette nouvelle branche en même temps — ça peut être réalisé en une seule opération avec :

```
git checkout -b <nouvelle-branche>.
```

Depuis Git version 2.23, on peut utiliser **`git switch`** au lieu de **`git checkout`** pour :

- Basculer sur une branche existante : **`git switch testing-branch`**,
- Créer une branche et basculer dessus : **`git switch -c nouvelle-branche`**;

  le drapeau `-c` signifie créer, vous pouvez aussi utiliser le drapeau complet `--create`,

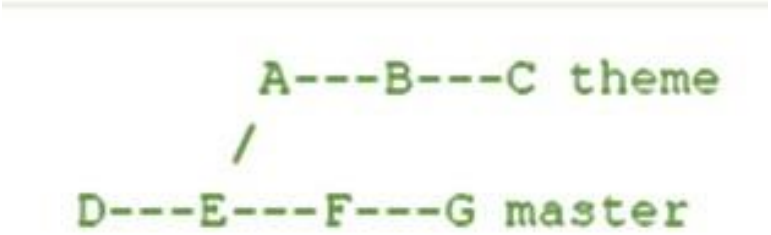
- *R*evenir sur votre branche précédemment extraite : **`git switch`**

Fusionner les branches

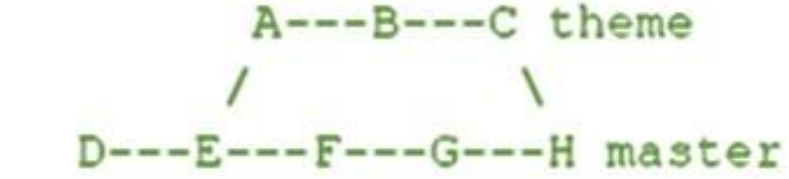
git-merge - Fusionne deux ou plusieurs historiques de développement ensemble

- Intègre les modifications des commits nommés (depuis le moment où leur historique a divergé de la branche actuelle) dans la branche actuelle.

Supposons que l'historique suivant existe et que la branche actuelle est master :



- La branche "**theme**" s'est écartée de **master** (c'est-à-dire E) jusqu'à son commit actuel (**C**);
- "**git merge theme**" rejouera les modifications apportées à la branche "**theme**" par dessus **master**;
- Elle enregistrera le résultat dans un nouveau commit comprenant les noms des deux parents et un message de validation de l'utilisateur décrivant les modifications.



git merge –abort: ne peut être exécutée qu'après que la fusion ait entraîné des conflits. **Elle** annulera le processus de fusion et tentera de reconstruire l'état antérieur à la fusion;

PARTIE 4

PARTIE 4

Conflits lors de fusion :

Un conflit de fusion intervient lorsque l'on tente de fusionner deux branches qui modifient la même partie d'un même fichier. Dans ce cas, git va intégrer les deux versions dans le même fichier puis laisser le développeur décider du contenu final de cette partie.

Exemple de conflit et sa résolution :

```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git
$ git init
Initialized empty Git repository in E:/Projects/projet_git/.git/

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ echo "ligne A" > file.txt

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git add .
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next time Git touches it

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git commit -m 'premier ajout du file.txt'
[master (root-commit) e9af967] premier ajout du file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git checkout –b fixbug
Switched to a new branch 'fixbug'
```

Exemple de conflit et sa résolution :

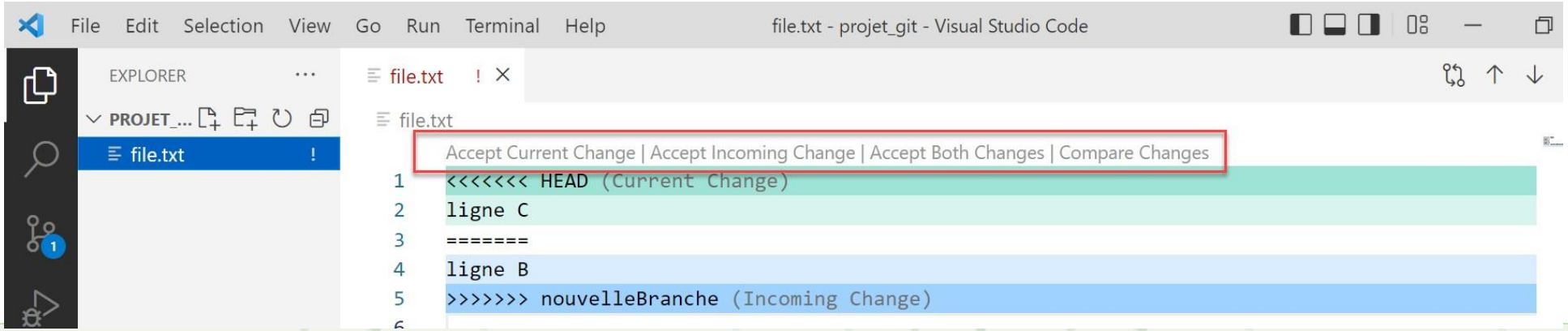
```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (fixbug)
$ echo "ligne B"  > file.txt

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (fixbug)
$ git commit -a -m 'Modification du file.txt'
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next time Git touches it
[fixbug 718cf8a] Modification du file.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git branch
  master
* fixbug

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (fixbug)
$ git checkout master
Switched to branch 'master'

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ echo "ligne C"  > file.txt
```

01-Manipuler les outils de gestion de versions (Git/Gitlab) :

Fusionner les branches

Exemple de conflit et sa résolution :

```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git commit -a -m 'Modification du file.txt à partir du master'
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next time Git touches it
[master c02d770] Modification du file.txt à partir du master
 1 file changed, 1 insertion(+), 1 deletion(-)

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git merge fixbug
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master|MERGING)
$ code .
```

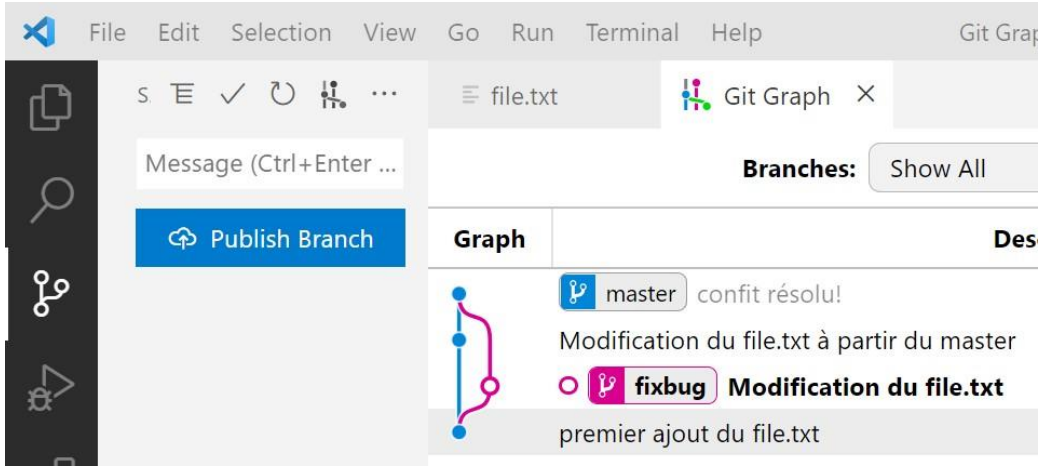


PARTIE 4

Exemple de conflit et sa résolution :

```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master|MERGING)
$ git commit -a -m 'confit résolu!'
[master 77f3c56] confit résolu!

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/projects/projet_git (master)
$ git log --oneline
77f3c56 (HEAD -> master) confit résolu!
c02d770 Modification du file.txt à partir du master
718cf8a (fixbug) Modification du file.txt
e9af967 premier ajout du file.txt
```

A l'aide de l'extension **git graph (sous visual studio code)**, on peut afficher graphiquement l'historique des commits et des branches



PARTIE 4

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. Intérêt de la gestion de version et présentation des outils existants de gestion de versions
2. Présentation de Git
3. **Présentation de Gitlab**
4. Manipulation des dépôts avec GitlabGestion des conflits de fusion avec Git/GitLab
5. Comparaison Git vs Gitlab

01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Présentation de Gitlab

C'est quoi Gitlab?

Gitlab est une plateforme open source et collaborative de développement basé sur **Git**. **Gitlab** permet d'héberger des projets web, du code, et de la documentation.

Fonctionnalités de Gitlab :

- L'interface de **GitLab** reste très similaire à celle de **GitHub**. Toutefois, **GitLab** propose des options pour le moins pratiques :
  - Gestion de projet
  - Planification / priorisation
  - Build
  - Test logiciel
  - Sécurité applicative
  - Gestion des configurations
  - Monitoring
  - Intégration et déploiement continus, etc.
- Pour un emploi ergonomique, **GitLab** se situe sur une machine virtuelle, elle-même hébergée sur un serveur web. Cet outil de plateforme collaborative s'appuie sur une base de données.
- L'interface d'administration, notamment pour la création de comptes utilisateurs, passe par une configuration en ligne :
  - Création / suppression de dépôts.
  - Enregistrement et gestion des droits d'accès aux dépôts .
  - Outils graphiques pour visualiser et éditer : graphe des commits, branches, tags, fichiers, etc.
  - Documentation des projets (fichiers README..)
  - Outils de communication de projets (issues)
  - Mécanisme de fork et merge requests (comme sur github).

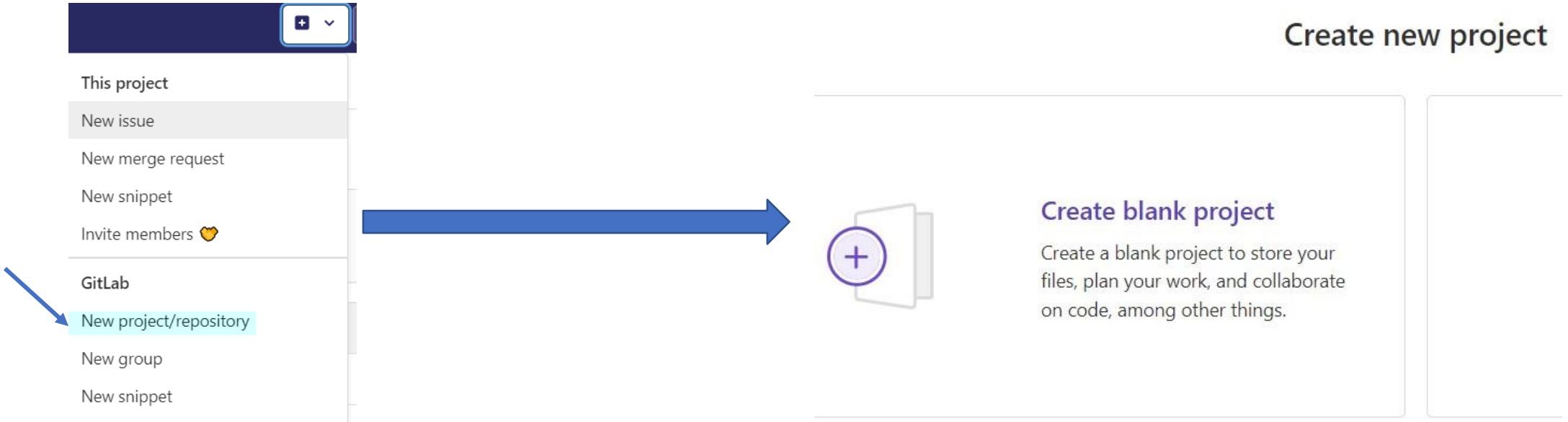
PARTIE 4

01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Création de compte

Etapes pour créer un compte sur Gitlab

- Ouvrir l'interface web **GitLab** sur : https://gitlab.com
- Si vous ne l'avez pas encore fait, ajoutez un mot de passe à votre compte **GitLab**, depuis la page de paramètres de votre compte **GitLab**
- Après la connexion avec le compte **Gitlab** vous aurez l'interface suivante :
  - Pour créer un projet cliquer sur **create blank project** ou **newProject/repository**



PARTIE 4

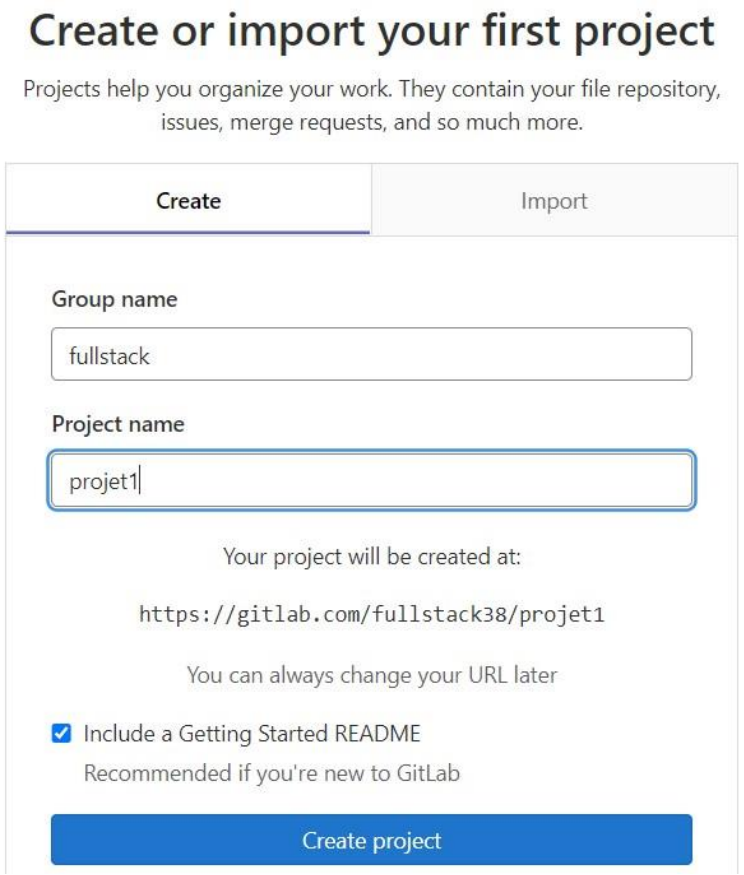
01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Création de compte

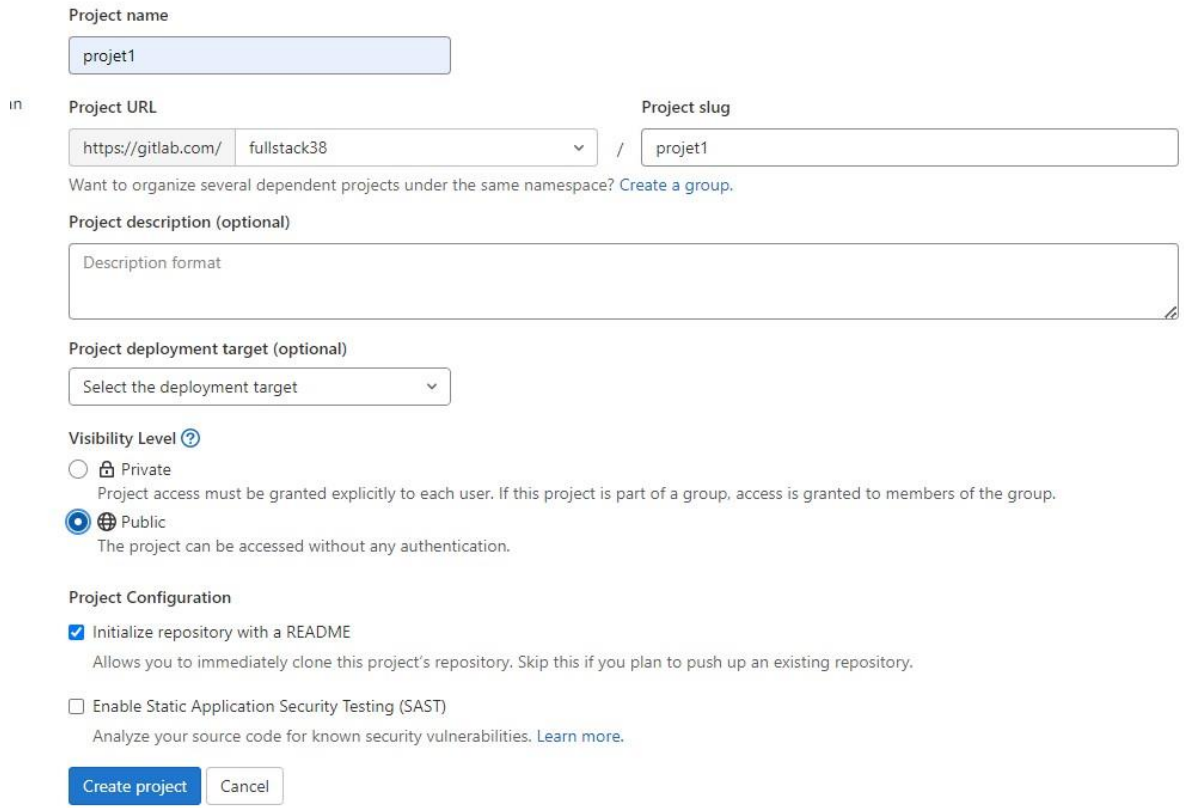
PARTIE 4

Etapes pour créer compte sur Gitlab

- Donner un nom à votre groupe et projet :



- Le nom du projet sera aussi le nom du dépôt correspondant => Choisissez “public”, afin que le dépôt soit accessible en lecture aux autres stagiaires, puis cliquer sur **create project**



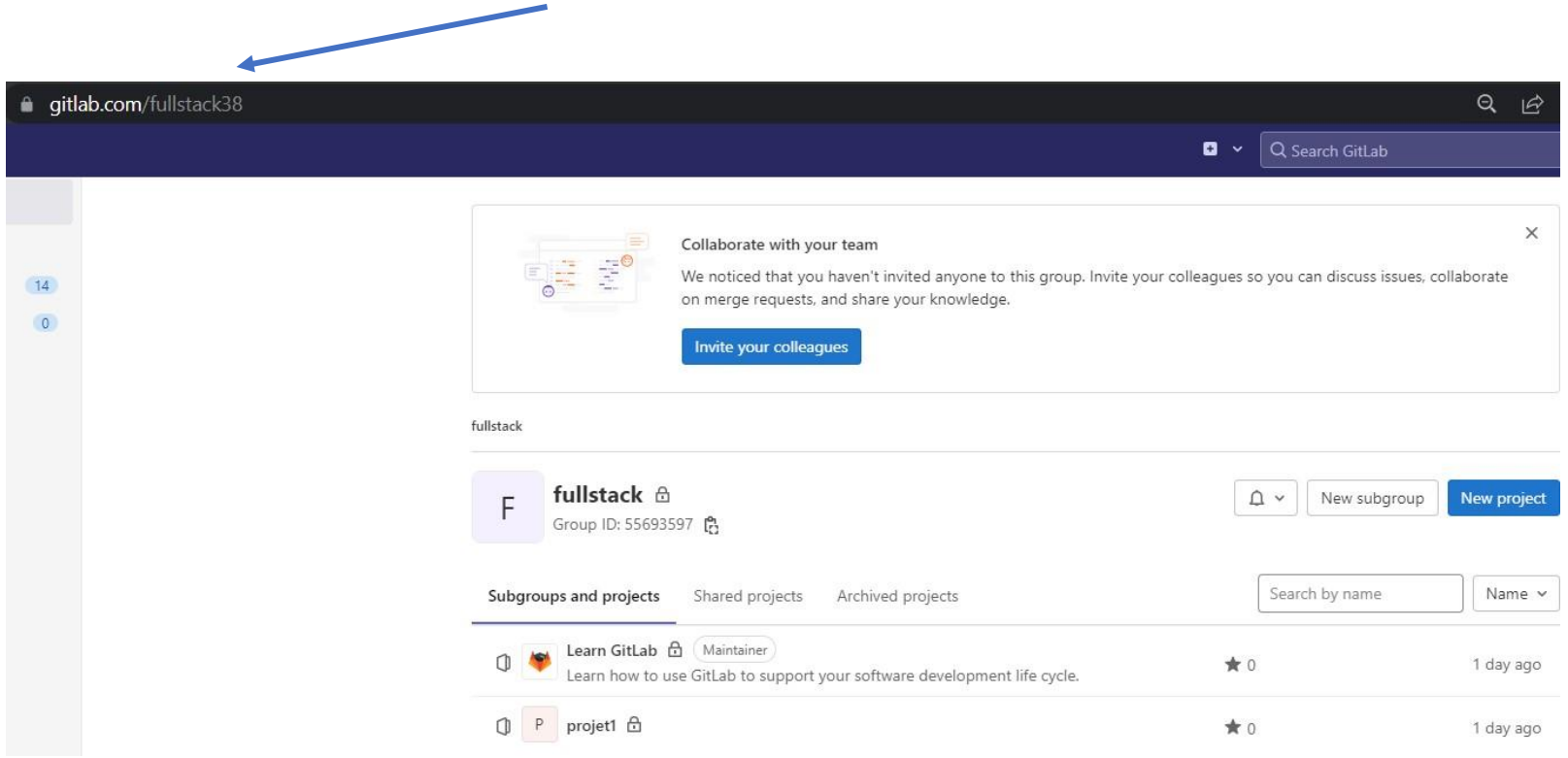
01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Création de compte

PARTIE 4

Etapes pour créer compte sur Gitlab

- Vous accéder à votre projet via l'url **: gitlab.com/groupeduprojet** :



CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. Intérêt de la gestion de version et présentation des outils existants de gestion de versions
2. Présentation de Git
3. Présentation de Gitlab
4. **Manipulation des dépôts avec Gitlab**
5. Gestion des conflits de fusion avec Git/GitLab
6. Comparaison Github vs Gitlab

Création dépôts sur Gitlab

1. Dans le terminal:
   - Téléchargez le dépôt sur votre disque-dur: **$ git clone <url> (remplacer <url> par l'adresse HTTPS de votre dépôt, celle qui finit par .git)**
   - puis entrez dans le dossier de ce dépôt: **$ cd nom_du_dépôt** ,exemple:

git clone https://gitlab.com/fullstack38/fsdev.git

```
C:\Users\DELL>git clone https://gitlab.com/fullstack38/fsdev.git
Cloning into 'fsdev'...
```

```
C:\Users\DELL>cd fsdev
```

PARTIE 4

Création dépôts sur Gitlab

2. Connecter à votre compte en utilisant les commandes de git suivantes avec votre **mail** et votre **username** :

```
DELL@DESKTOP-01DOJ3U MINGW64 ~/fsdev (main)
$  git config --global user.email "[redacted]@gmail.com"

DELL@DESKTOP-01DOJ3U MINGW64 ~/fsdev (main)
$  git config --global user.name "Mohamed[redacted]"
```

3. Aussi ,dans le terminal exécuter les commandes suivantes pour modifier votre projet

- Créer un commit:
  - **echo "Bonjour " >README.md** : pour créer un fichier README.md contenant le texte “Bonjour”
  - **git status :** (optionnel) pour constater qu’un fichier a été créé mais pas encore ajouté dans le dépôt
  - **git add README.md :**pour ajouter le fichier README.md dans l’espace de staging (index)
  - **git status :**(optionnel) pour afficher le contenu actuel de l’espace de staging (index)
  - **git commit -m "ajout du fichier README.md" :** pour créer un commit à partir de l’espace de staging. Notez que le texte fourni entre guillemets est libre.
  - **git status :**(optionnel) pour constater que l’espace de staging a été réinitialisé et qu’aucun fichier n’a été modifié depuis votre commit
  - **git push :** pour uploader votre commit sur votre dépôt distant, hébergé sur le serveur GitLab.

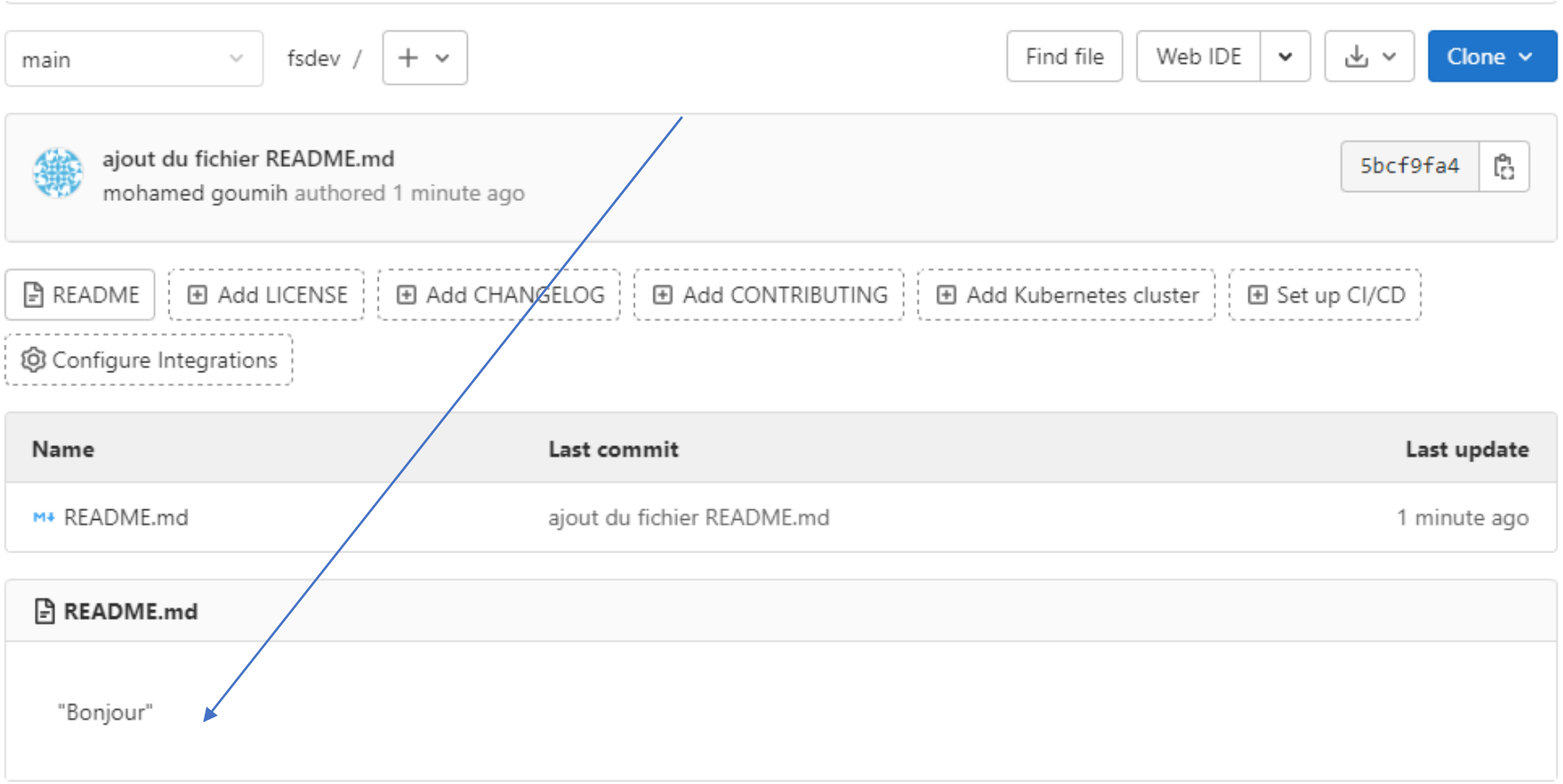
Remarque : Ses commandes à exécuter lorsque vous voulez créer un nouveau fichier dans le projet

PARTIE 4

PARTIE 4

Création dépôts sur Gitlab

4. Le fichier **README.md** devrait maintenant être visible depuis la page web du dépôt, sur **GitLab** :



01-Manipuler les outils de gestion de versions(Git/Gitlab) :

Gestion branches avec git

Gestion branches avec commandes git

créer une nouvelle branche nommée B1 et passer dessus pour l'utiliser
git checkout -b B1
retourner sur la branche principale
git checkout master
et supprimer la branche
git branch -d B1
une branche n'est *pas disponible pour les autres* tant que vous ne l'aurez pas envoyée vers votre dépôt distant
git push origin <branch>

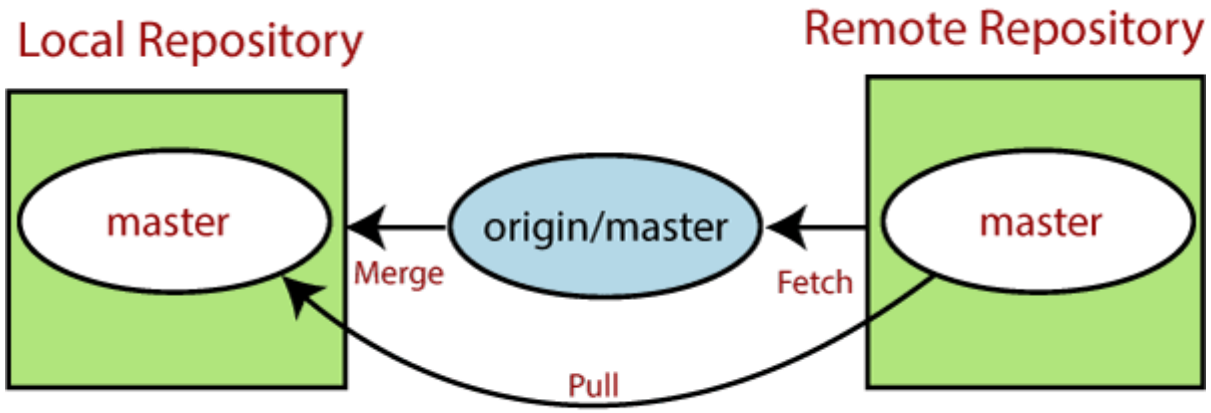
pour mettre à jour votre dépôt local depuis les dernières validations, exécutez la commande
git pull
dans votre espace de travail pour *récupérer* et *fusionner* les changements distants.
pour fusionner une autre branche avec la branche active (par exemple master), utilisez
git merge <branch>
dans les deux cas, git tente d'auto-fusionner les changements. Malheureusement, ça n'est pas toujours possible et résulte par des *conflits*. Vous devez alors régler ces *conflits* manuellement en éditant les fichiers indiqués par git. Après l'avoir fait, vous devez les marquer comme fusionnés avec
git add <nomdufichier>
après avoir fusionné les changements, vous pouvez en avoir un aperçu en utilisant
git diff <source_branch> <target_branch>

vous pouvez annuler les changements locaux en utilisant cette commande
git checkout -- <nomdufichier>
cela remplacera les changements dans votre arbre de travail avec le dernier contenu du HEAD. Les changements déjà ajoutés à l'index, aussi bien les nouveaux fichiers, seront gardés.
Si à la place vous voulez supprimer tous les changements et validations locaux, récupérez le dernier historique depuis le serveur et pointez la branche principale locale dessus comme ceci
git fetch origin
git reset --hard origin/master

PARTIE 4

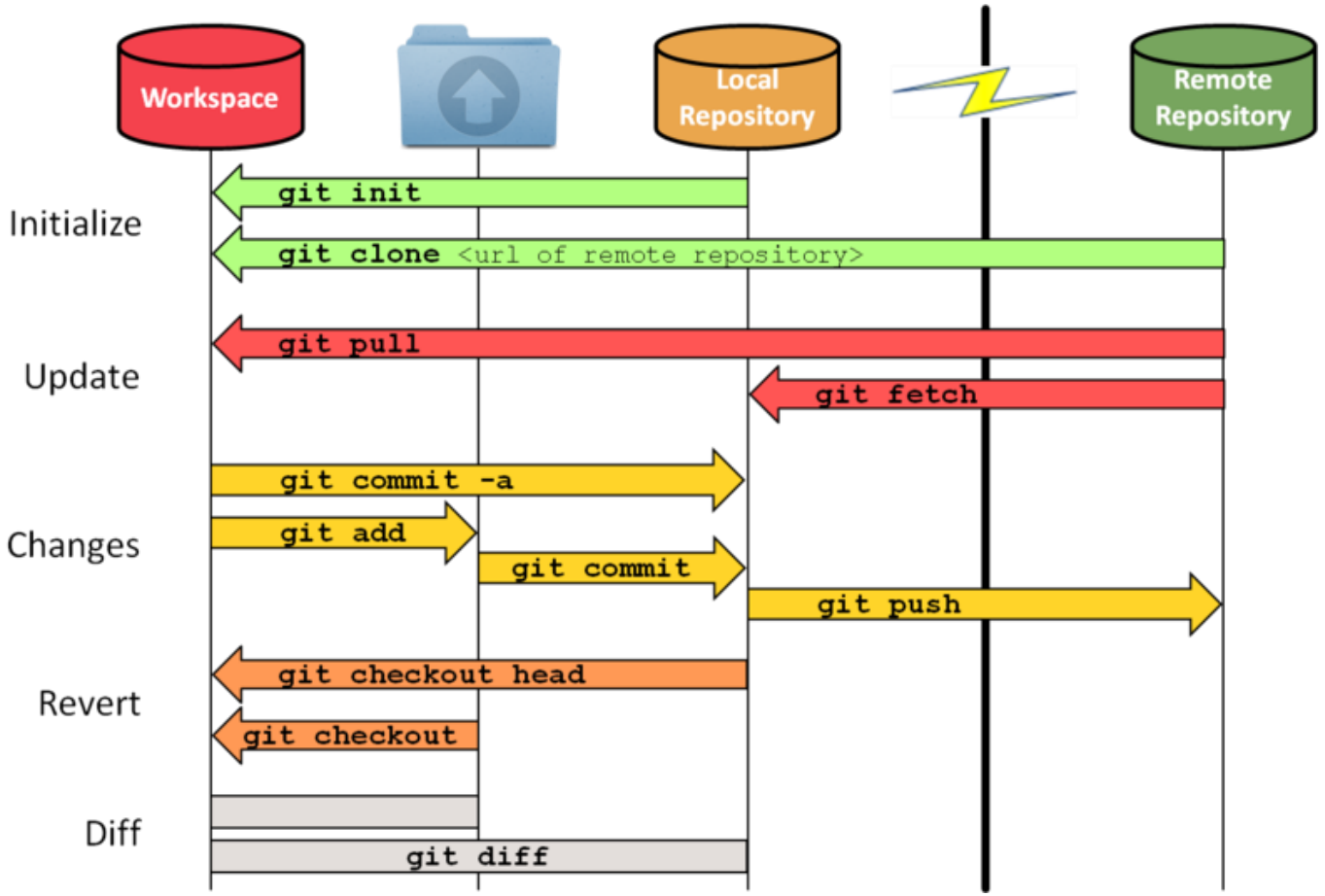
La commande git fetch:

La commande git fetch permet de récupérer les modifications présentes sur le serveur distant que vous n'avez pas encore sur votre copie en local. Cette commande va alors mettre à jour dans votre dépôt local, l'ensemble des références distantes (branches, tags, …) et récupérer les commits associés.



PARTIE 4

Récapitulatif des commandes git



Git tag

- Les **tag** sont des références qui pointent vers des points spécifiques de l'historique de Git.
- Le **tag** est généralement utilisé pour capturer un point dans l'historique qui est utilisé pour une version release (exp. : v1.0.1).
- Un tag est une branche qui ne change pas.
- Contrairement aux branches, les tag ne gardent plus d'historique des prochains commits.

Créer un tag	**git tag** <tagname>	git tag v1.0
Créer un tag annoté : stocke des informations supplémentaires concernant la personne qui a crée le tag, la date …	**git tag -a** <tagname> **-m** <message>	git tag -a v1.4 - m "my version 1.4"
Lister les tags	**git tag**	
Partager un tag avec le repertoire distant	**git push** <remote> <tagname>	git push origin v1.4
Supprimer un tag	**git tag –d** <tagname>	git tag -d v1.4

PARTIE 4

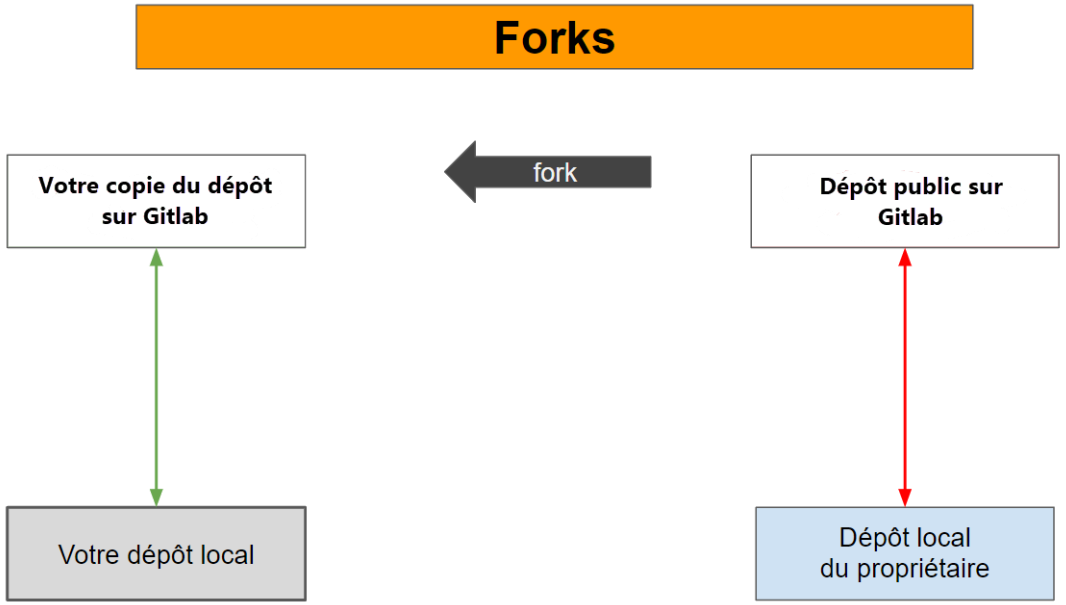
PARTIE 4

Le fork avec Gitlab : Contribution à un projet

- Si vous souhaitez contribuer à un projet existant sur lequel vous n'avez pas le droit de pousser, vous pouvez dupliquer (fork) ce projet. Cela signifie que Gitlab va faire pour vous une copie personnelle du projet. Elle se situe dans votre espace de nom et vous pouvez pousser dessus.

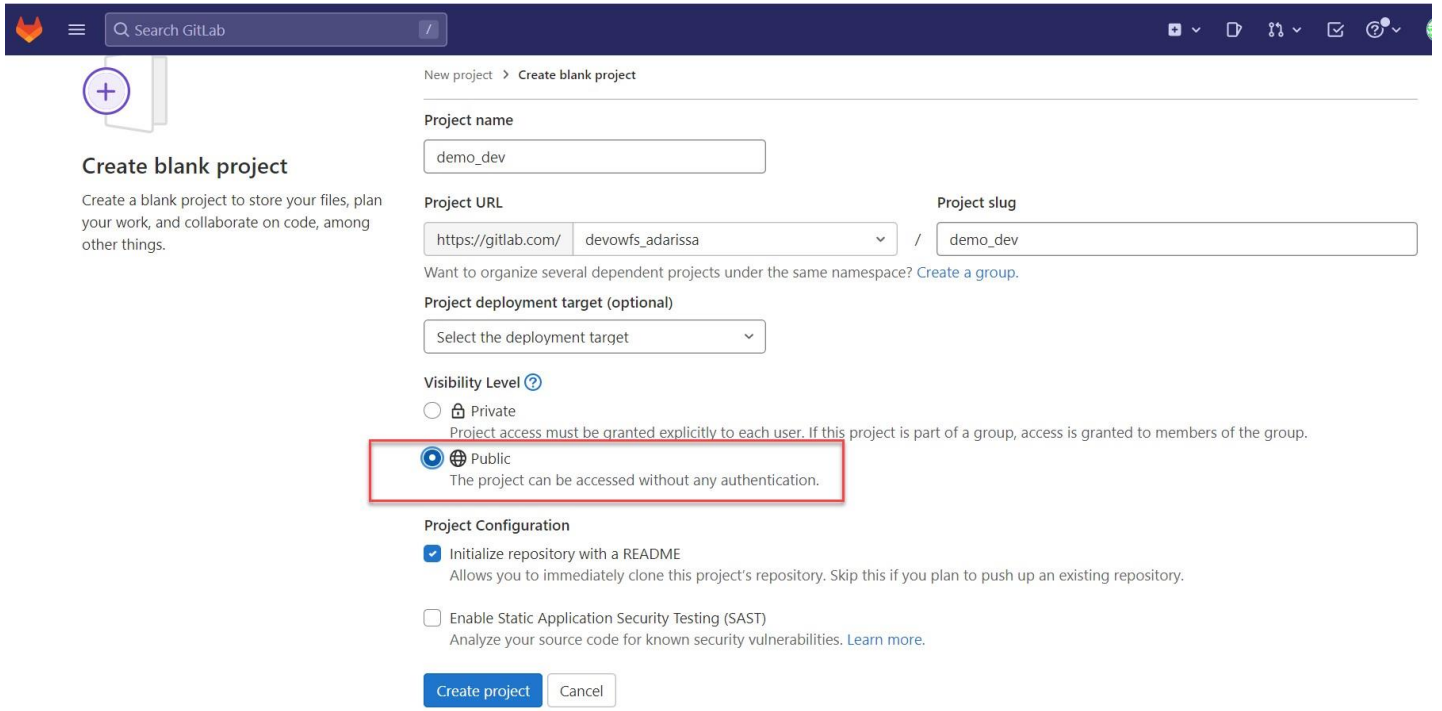
- **Un fork :** est une copie d'un dépôt. Ceci est utile lorsque vous souhaitez contribuer au projet de quelqu'un d'autre ou démarrer votre propre projet basé sur le sien.

- **fork** n'est pas une commande dans Git, mais quelque chose d'offert dans GitLab et d'autres référence de versions.



Le fork avec Gitlab : Contribution à un projet

- Créons un nouveau projet public ;
- Le projet sera accessible par tout le monde et sans aucune authentification;
- N'importe qui peut cloner le projet pour en créer une copie local ;



- **Pourtant**; l'erreur suivante sera affichée si on essaie de faire un push sur ce projet public !

(Après l'avoir cloner)

```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/js2 (master)
$ git push
remote: Permission to [redacted] denied to [redacted]
la.
fatal: unable to access 'https://git[redacted].g
it/': The requested URL returned error: 403
```

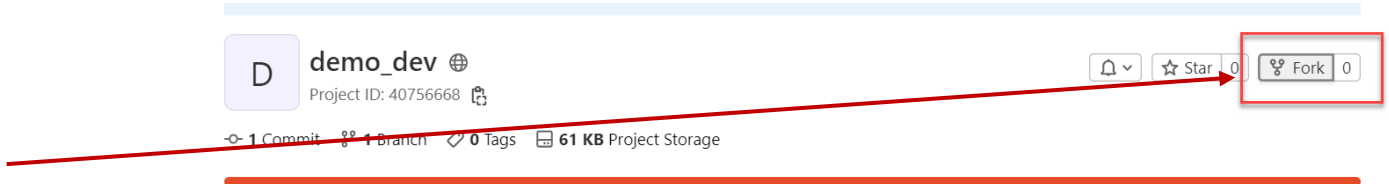
PARTIE 4

PARTIE 4

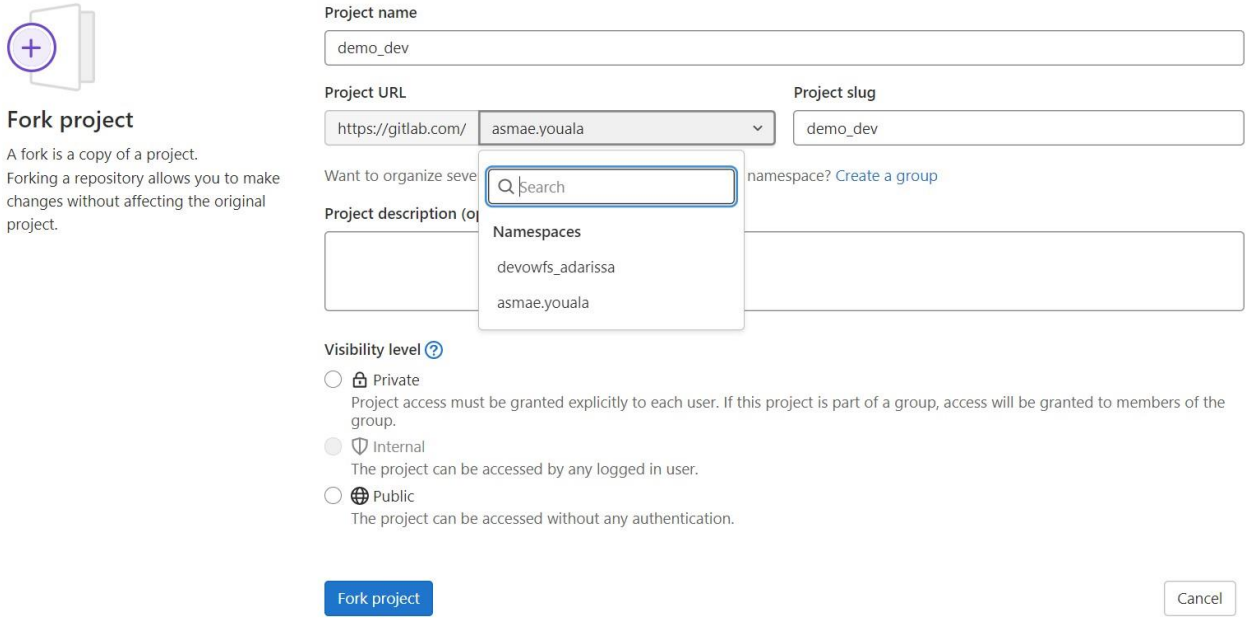
Le fork avec Gitlab : Contribution à un projet

Afin de pouvoir contribuer à un projet distant, on peut utiliser l'option fork de gitlab:

- En cliquant sur fork, la nouvelle fenêtre « fork project » apparait;



- On peut indiquer, sur cette page, le nom de notre fork, son namespace, sa visibilité …



01-Manipuler les outils de gestion de versions (Git/Gitlab) :

Manipulation des dépôts avec Gitlab

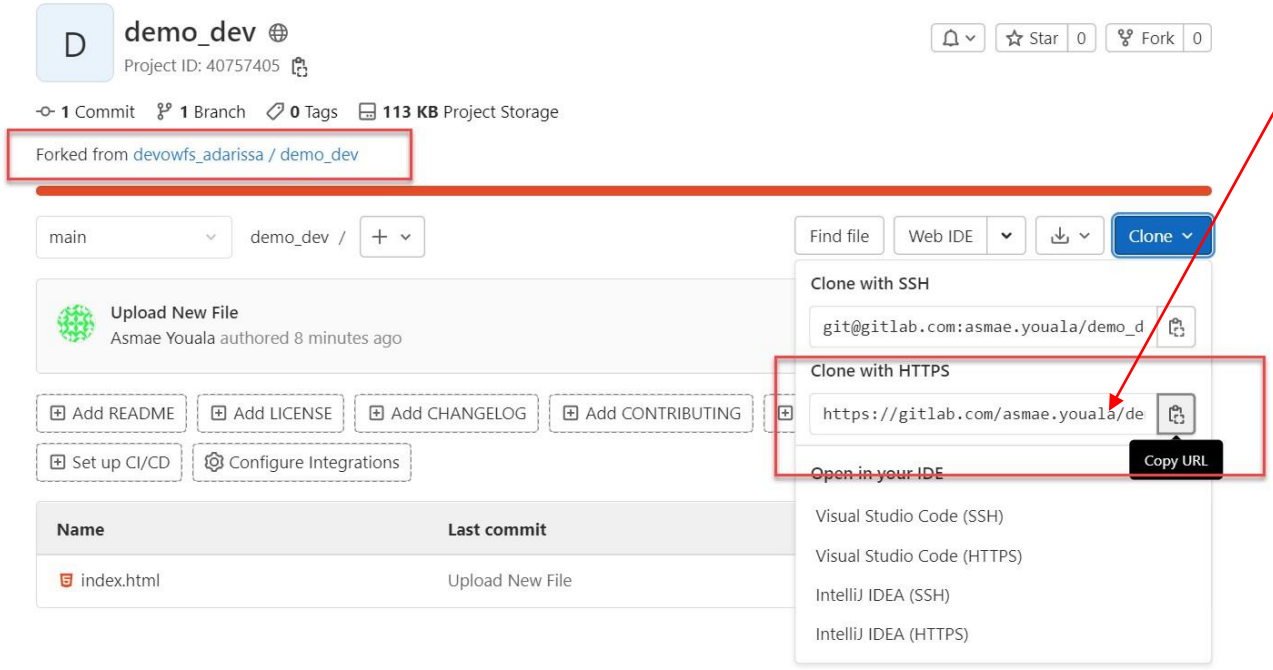
Le fork avec Gitlab : Contribution à un projet

Sur le répertoire de travail, créons un accès à notre fork en lançant la commande suivante:

```
$ git remote add ma_copie https://gitlab.com/asmae.youala/demo_dev.git
```

Un nom de mon choix

L'adresse du projet copié (forked project)



Le fork avec Gitlab : Contribution à un projet

Afin d'afficher les répertoires à distance du projet (cloned project + forked project), utilisez la commande suivante :

```
$ git remote -v
```

```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/demo_dev (main)
$ git remote -v
ma_copie        https://gitlab.com/asmae.youala/demo_dev.git (fetch)
ma_copie        https://gitlab.com/asmae.youala/demo_dev.git (push)
origin  https://gitlab.com/devowfs_adarissa/demo_dev (fetch)
origin  https://gitlab.com/devowfs_adarissa/demo_dev (push)
```

origin: c'est le projet original en lecture seule
ma_copie: c'est mon fork avec un accès en lecture et en écriture

Pour pousser les modifications vers le projet fork, il suffit de lancer la commande :

```
$ git push ma_copie
```

Nous pouvons renommer l'un des répertoires en utilisant la commande :

```
$ git remote rename ancien_nom nouveau_nom
```

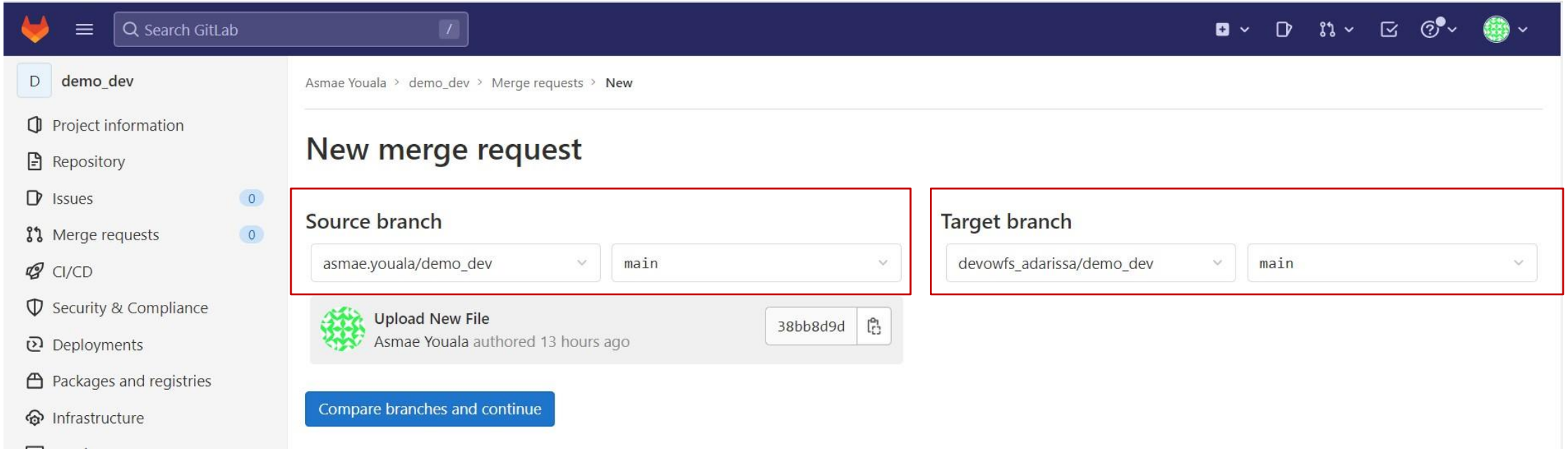
```
asmae@DESKTOP-PGQ5OJJ MINGW64 /e/demo_dev (main)
$ git remote rename ma_copie fullstack

asmae@DESKTOP-PGQ5OJJ MINGW64 /e/demo_dev (main)
$ git remote -v
fullstack       https://gitlab.com/asmae.youala/demo_dev.git (fetch)
fullstack       https://gitlab.com/asmae.youala/demo_dev.git (push)
origin  https://gitlab.com/devowfs_adarissa/demo_dev (fetch)
origin  https://gitlab.com/devowfs_adarissa/demo_dev (push)
```

PARTIE 4

PARTIE 4

Merge Request avec Gitlab

- **Merge Request** (Les demandes de fusion) : (**Pull Request** en github): sont la façon dont vous vérifiez les modifications du code source dans une branche. Lorsque vous ouvrez une demande de fusion, vous pouvez visualiser et collaborer sur les changements de code avant la fusion.
- Vous pouvez créer une Merge Request à partir de la liste des demandes de fusion.
- Sur le menu gauche
    1. Dans le menu de gauche de votre projet (forked), sélectionnez **Merge Requests**.
    2. puis, sélectionnez **new merge request**.
    3. Sélectionnez une branche source et cible, puis Comparez les branches et continuez.
    4. Remplissez les champs et sélectionnez **create merge request**.



Merge Request avec Git/gitlab

- Vous pouvez créer une Merge Request en exécutant des commandes Git sur votre ordinateur local.

1. Créer une branche:
   git checkout -b my-new-branch
2. Créer ou modifier vous fichiers ,puis les met dans le stage et les valider :
   git add .
   git commit -m " message de valdation"
3. Poussez votre branche vers GitLab :
   git push –u origin my-new-branch
4. GitLab vous invite avec un lien direct pour créer une demande de fusion :
   Copiez le lien et collez-le dans votre navigateur.

01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Manipulation des dépôts avec Gitlab

Collaborer sur un dépôt Gitlab

- Par défaut seul le créateur d'un dépôt peut y ajouter des **commits**.
- Si un autre développeur souhaite contribuer au dépôt, il existe deux manières de procéder:
  - ***forker*** le dépôt puis proposer une contribution à l'aide d'un ***pull request(merge request)***;
  - ou obtenir la permission d'ajouter des **commits** directement dans le dépôt.

Etapes pour ajouter un développeur dans une équipe Gitlab:

1. Depuis la barre latérale de la page du dépôt, cliquer sur "**Paramètres**",
2. Cliquer sur "Membres",
3. Taper le ou les noms d'utilisateurs (ou adresse email) du/des développeurs à ajouter,
4. Sélectionner le rôle (ou niveau de permissions) à donner à ce(s) développeur(s),
5. Vérifier que le(s) développeur(s) est/sont bien capables d'ajouter et pusher un commit dans la branche de travail du dépôt (ex: master).

Note: pour qu'un développeur aie le droit de pusher des **commits** dans votre dépôt, il faut lui donner le rôle de "**Maintainer**". Le rôle "**Developer**" ne suffit pas.

Etapes pour ajouter un commit dans le dépôt d'un autre développeur

1. Utiliser git clone pour importer le dépôt de l'autre stagiaire
2. S'assurer qu'on a bien les dernières mises à jour (git pull)
3. Créer un nouveau fichier dans le dépôt local, puis l'ajouter à l'index (git add)
4. Créer un commit contenant ce fichier (git commit) puis l'envoyer sur le dépôt de l'autre stagiaire (git push)
5. Dans l'interface web de GitLab, aller sur le projet de votre camarade, puis cliquez sur "commits" pour vérifier que votre commit apparaît bien dans la liste.

PARTIE 4

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. Intérêt de la gestion de version et présentation des outils existants de gestion de versions
2. Présentation de Git
3. Présentation de Gitlab
4. Manipulation des dépôts avec Gitlab
5. **Gestion des conflits de fusion avec Git/GitLab**
6. Comparaison Github vs Gitlab

01- Manipuler les outils de gestion de versions (Git/Gitlab) :

Gestion conflits avec Gitlab

Définition d'un conflit de fusion

Un conflit de fusion intervient lorsque l'on tente de fusionner deux branches qui modifient la même partie d'un même fichier. Dans ce cas, git va intégrer les deux versions dans le même fichier puis laisser le développeur décider du contenu final de cette partie.

Etapes à suivre pour causer un conflit de fusion :

1. Créer un nouveau dépôt sur **GitLab**
2. Cloner ce dépôt localement avec git clone
3. Dans le répertoire du dépôt local, créer un fichier **README.md** contenant un texte
4. Créer un commit initial sur la banche master et l'envoyer sur **GitLab** avec **git add, git commit** puis **git push**
5. Créer une branche branche1 à partir de master, avec git **checkout -b**
6. Dans le **README.md** de cette branche, modifier le premier texte, créer un commit, puis envoyer les modifications de cette branche sur **GitLab**
7. Revenir à la branche master avec **git checkout**
8. Créer une branche **branche2** à partir de master (comme dans l'étape 5)
9. Dans le **README.md** de cette branche, modifier le texte différemment de celui saisi à l'étape 6, créer un commit, puis envoyer les modifications de cette branche sur **GitLab**
10. Revenir à **la branche master**
11. Fusionner **branche1** dans master, avec **git merge**
12. Fusionner **branche2** dans master
13. Taper **git status** pour voir le message du conflit

Etapes à suivre pour résoudre un conflit

Pour résoudre ce conflit, il va falloir:

1. ouvrir le fichier **README.md** dans son éditeur de code,
2. constater comment git représente le conflit, et la source de chaque version,
3. éditer le fichier pour ne conserver que la version finale souhaitée,
4. puis créer un commit.
5. Après la résolution du conflit de fusion, taper **git status** pour voir le résultat
6. supprimer les branches **branche1** et **branche2**, dans votre dépôt local, et dans le dépôt distant associé (sur **GitLab**).

PARTIE 4

CHAPITRE 1

Manipuler les outils de gestion de versions (Git/Gitlab)

1. Intérêt de la gestion de version et présentation des outils existants de gestion de versions
2. Présentation de Git
3. Présentation de Gitlab
4. Manipulation des dépôts avec Gitlab
5. Gestion des conflits de fusion avec Git/GitLab
6. **Comparaison Github vs Gitlab**

Quelques Fonctionnalités semblables entre gitlab et github

GitHub et **GitLab** reposent tous les deux sur **Git** et ses commandes, la **migration** d'une plateforme à l'autre est possible sans grandes difficultés. L'importation des repositories, **wikis**, **pull requests** et **issues** est facile en règle générale. En revanche, il existe quelques différences entre **GitHub** et **GitLab**, comme l'illustre le tableau ci-dessous :

Fonctionnalité	GitLab	GitHub
Git	✓	✓
Version auto-hébergée	✓	✓ (avec plan d'entreprise)
Intégration et livraison continues	✓	✓ (avec une application tierce)
Documentation basée sur le Wiki	✓	✓
Aperçu des modifications du code	✓	✓
Suivi des problèmes	✓	✓
Examen du code	✓	✓
Cessionnaires d'émissions multiples	✓ (Plan payant)	✓ (uniquement le dépôt public sur le plan gratuit)
Conseils de gestion de projet	✓	✓
Discussions d'équipe	✓	✓
Suivi du temps	✓	✓ (Avec App)
Outils de sécurité et de conformité	✓	✓
Test de performance de charge	✓ (Plan payant)	✓ (Avec App)
Test de performance des navigateurs	✓ (Plan payant)	✓ (Avec App)
Itérations et planification des sprints (y compris Burndown Chart)	✓ (Plan payant)	✓ (Avec App)
Dépendances des problèmes	✓ (Plan payant)	✓

01- Manipuler les outils de gestion de versions

Git/Gitlab :

Comparaison Github et Gitlab

Quelques différences majeures entre gitlab et gitlab

GitHub	GitLab
Les issues peuvent être suivies dans plusieurs repositories	Les issues ne peuvent pas être suivies dans plusieurs repositories
Repositories privés payants	Repositories privés gratuits
Pas d'hébergement gratuit sur un serveur privé	Hébergement gratuit possible sur un serveur privé
Intégration continue uniquement avec des outils tiers (Travis CI, CircleCI, etc.)	Intégration continue gratuite incluse
Aucune plateforme de déploiement intégrée	Déploiement logiciel avec Kubernetes
Suivi détaillé des commentaires	Pas de suivi des commentaires
Impossible d'exporter les issues au format CSV	Exportation possible des issues au format CSV par e-mail
Tableau de bord personnel pour suivre les issues et pull requests	Tableau de bord analytique pour planifier et surveiller le projet
Interface graphique un peu compliqué	Interface graphique clair et facile