Introduction à git/gitlab (avec une seule branche) et maven

Git et gitlab

- Nous utiliserons gitlab.com pour le dépôt distant, il vous faut donc un compte gitlab.com. En effet, nous aurons besoin de fonctionnalités qui ne sont pas dans le gitlab de l'UM, et qui existent mais sous des formes sur les autres plateformes Git comme GitHub. Il vous faut donc impérativement un compte gitlab.com.

- Si ce n'est pas déjà fait (dans une autre vie), placez une clef publique dans votre compte gitlab. Cela permettra de vous identifier notamment lorsque vous poussez du code vers votre dépôt distant. Pour cela, suivez attentivement la procédure indiquée ici, et n'oubliez pas votre passphrase.

Principes de base de git (rappel)

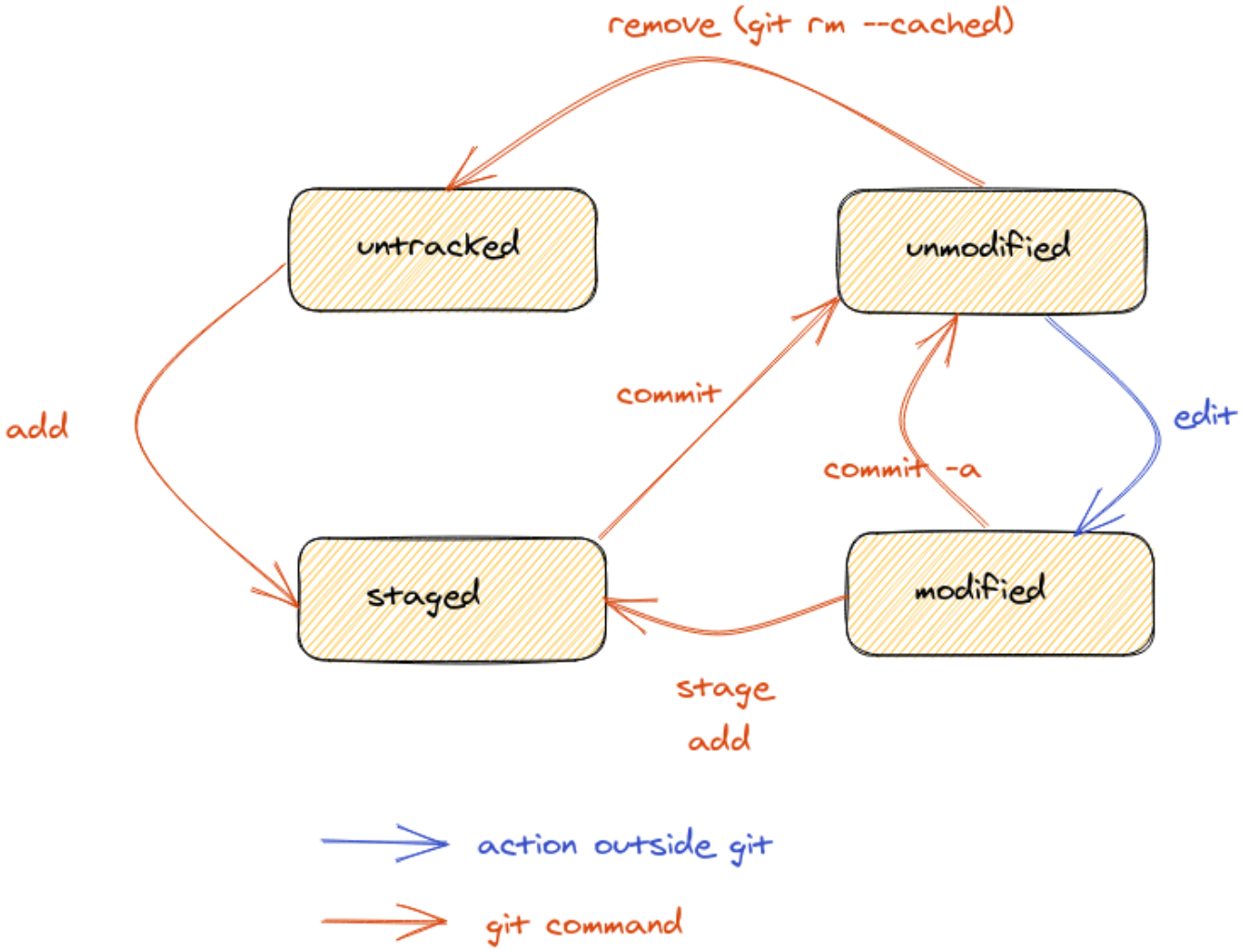
Git fonctionne en travaillant sur 4 zones différentes, comme illustré ci-dessous.

- Le répertoire de travail : comme son nom l'indique, le répertoire de travail est celui où le développeur travaille.
- la zone de staging : cette zone sert en quelque sorte de tampon entre le répertoire de travail et dépôt local. Elle sert à préparer les modifications (à archiver/historiciser) des fichiers suivis.
- le dépôt local : cette zone stocke localement l'historique des modifications apportées aux fichiers suivis. Ces modifications peuvent être locales comme émanant d'autres programmeurs.
- le dépôt distant : cette zone sera pour nous sur gitlab.com. Elle sert à partager les modifications de tous les programmeurs.

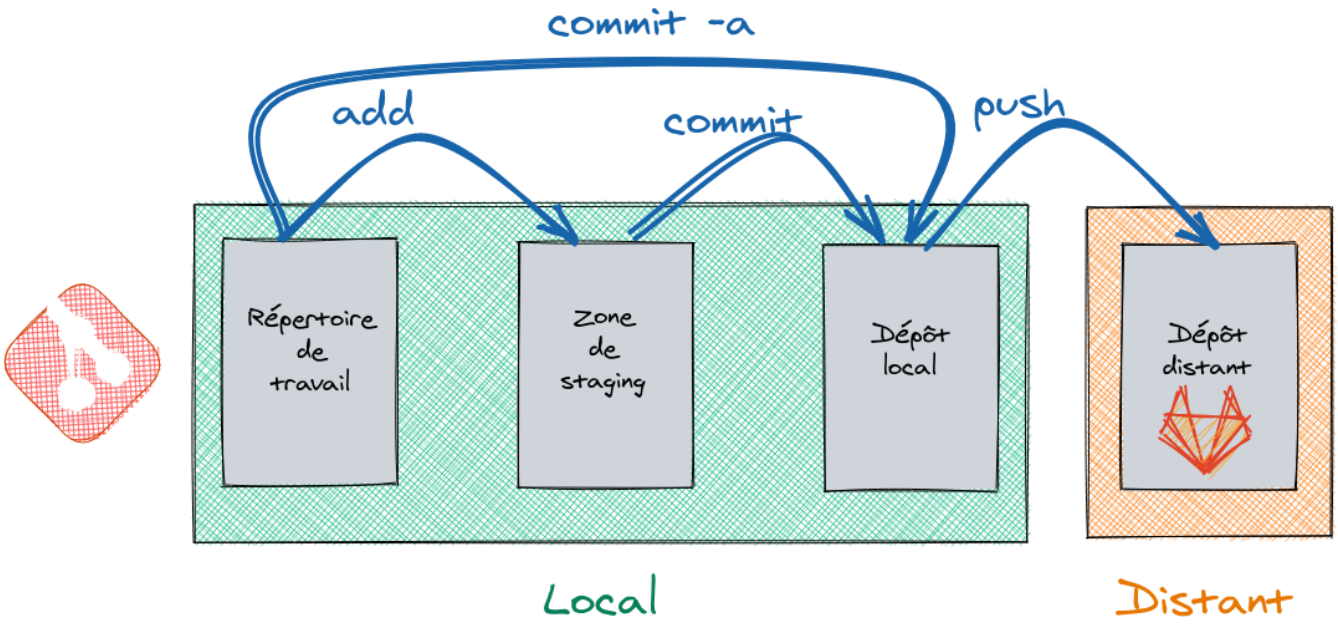
Un même fichier du répertoire de travail peut donc être dans différents états. A gros grains, le fichier est soit tracked (suivi par git), soit untracked (non suivi par git). A grain plus fin, il y a 4 états car l'état tracked se découpe en 3 sous-états :

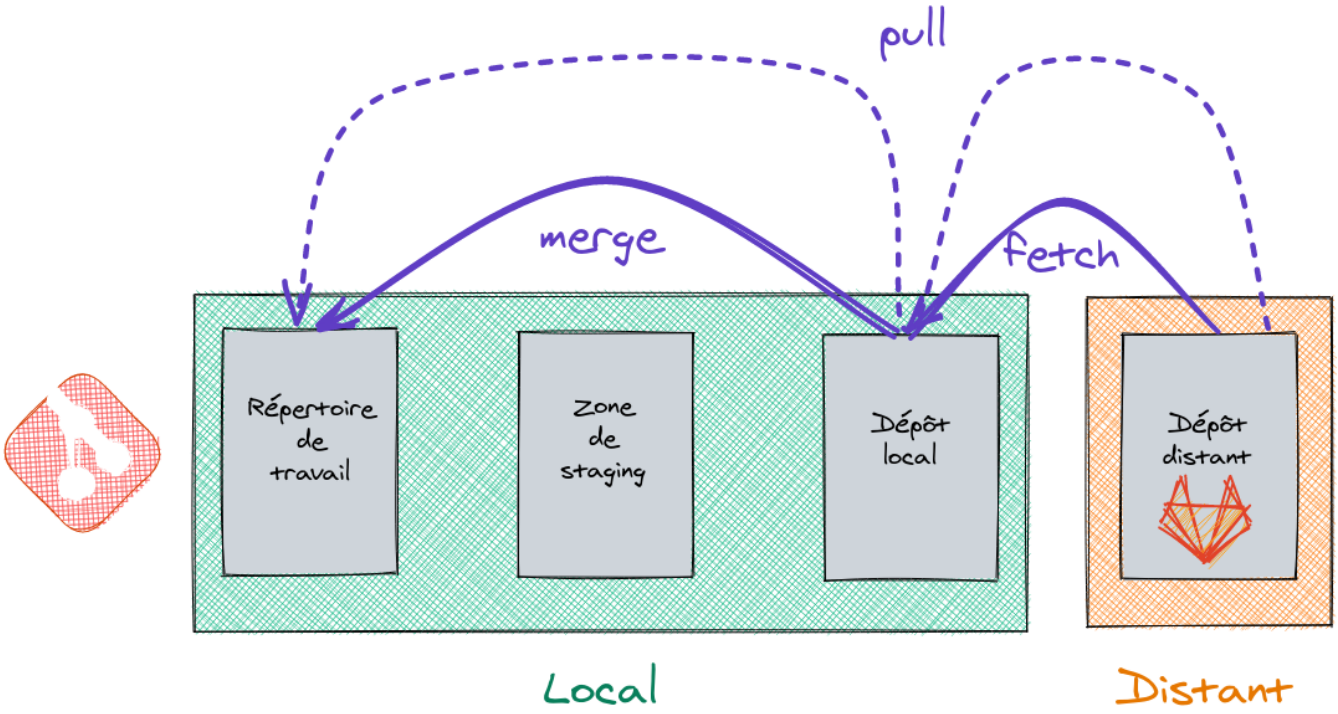
- untracked : le fichier est dans le répertoire de travail mais n'est pas suivi par git, il ne sera pas historicisé
- modified : le fichier est suivi par git, et est modifié dans le répertoire de travail par rapport au dépôt local.
- staged : le fichier sera archivé lors du prochain commit.
- unmodified : le fichier n'est pas modifié dans le répertoire de travail : il est donc à jour par rapport au dépôt local

Différentes commandes permettent de passer d'un état à un autre (et donc d'être à jour ou pas avec le dépôt local) comme illustré à la figure suivante.



Commandes git de base





Mise en pratique et travail collaboratif (interface gitlab+ligne de commande)

à faire tout seul

- Créez un projet sur gitlab.com, nommé gitlab1. Placez-y (depuis l'interface de gitlab.com) un fichier README.md de contenu :

  ```
  # Projet gitlab pour tester les premières manipulations git/gitlab
  ```

  Vous utiliserez pour cela le bouton avec un + permettant d'ajouter des éléments dans votre projet depuis l'interface gitlab.

- Dans votre homedir, placez-vous dans le répertoire dans lequel vous souhaitez travailler, que l'on nommera ici TP-AGL mais que vous pouvez appeler comme vous voulez.

- clonez le projet gitlab1 : `git clone <projet>` où <projet> est l'adresse de votre dépôt que vous pouvez récupérer sur la page de votre projet. ATTENTION : l'adresse à utiliser ici est celle prévue pour une communication en SSH, pas en http !

- Vous avez maintenant un répertoire de travail, dont la branche `origin` référence votre projet sur gitlab.com.

- Depuis le répertoire gitlab1, faites un `git status` : le répertoire de travail est à jour.

- Ajoutez à TP-AGL/gitlab1 un fichier `HelloWorld.java` de contenu :

  ```
  public class HelloWorld{
      public static void main (String[] args){
          System.out.println("Hello, world !");
  ```

```
    }
}
```

- Refaîtes un `git status` : cela vous indique que le nouveau fichier `HelloWorld.java` n'est pas suivi.
- On suit maintenant le fichier `HelloWorld.java` : `git add HelloWorld.java`.
- De nouveau `git status`. Dans les "Modifications qui seront validées" on trouve maintenant notre nouveau fichier, qui est "indexé" c'est-à-dire dans l'état "staged".
- Faîtes un `git rm --cached HelloWorld.java` ou un `git restore --staged HelloWorld.java`, puis `git status` : le fichier redevient non suivi.
- Recommencez le `git add HelloWorld.java`.
- Puis `git commit`. Gargl ! Git vous demande un message expliquant votre commit, et il a ouvert (en général et selon les configurations) l'éditeur `vi` !.
    - Si vous maîtrisez `vi` ou qu'un éditeur que vous maîtrisez s'est ouvert, tapez votre message, sauvez, quittez.
    - Sinon quittez. Le commit n'est pas validé car le message était vide. Deux options : soit passer le message en ligne de commande avec un `git commit -m "ajout du fichier HelloWorld.java"` soit configurer git pour qu'il ouvre un autre éditeur. Pour cela, git config --global core.editor "nano" permet par exemple de choisir nano.
- De nouveau `git status`. La copie de travail est "propre".
- Modifiez `README.md`. Puis `git add README.md` pour indexer les modifications.
- Ensuite modifiez `HelloWorld.java` mais ne l'indexez pas.
- De nouveau `git status`. Il vous est indiqué plusieurs choses.
    - Votre branche est en avance sur 'origin/main' de 1 commit signifie que puisque vous n'avez pas encore poussé vos modifications locales vers le dépôt distant, votre branche de travail est en avance de 1 commit.
    - Lors du prochain commit les modifications du fichier `README.md` seront validées mais pas celles du fichier `HelloWorld.java`.
- Faîtes un `git commit` puis un `git push` et un `git status`. Vous êtes de nouveau à jour avec la branche main et vous voyez dans l'interface gitlab vos fichiers.
- Dans votre répertoire de travail, compilez `HelloWorld.java`. Vous produisez donc un `.class`. Dans le git on ne veut pas faire apparaître les fichiers compilés. Mais on voudrait éviter que git les considère (en nous informant qu'ils ne sont pas suivis par exemple).
- Pour que git ignore des fichiers, on lui donne une liste de patterns décrivant les fichiers à ignorer :
    - si ces patterns sont communs à tout le projet on les place à la racine du dossier de travail, et ce fichier sera archivé dans le dépôt local puis distant.

    - si ces patterns sont communs à tous les projets d'un utilisateur, on place un fichier dédié dans la configuration de git de cet utilisateur.
    - si ces patterns sont propres à un utilisateur et un projet, on place le fichier de patterns dans le répertoire `info/exclude` du projet.

- On se place ici dans le cas 1. Pour cela, nous allons faire un `.gitignore` à placer dans votre répertoire de travail, et à indexer. Le contenu de ce fichier pourrait être :

```
# exclusion des .class
*.class
```

- Un `git status` ne mentionne pas le fichier compilé (`HelloWorld.class`) et un `git status --ignored` montre bien que git a ignoré ce fichier.
- Ajoutez, commitez et poussez toutes les modifications de votre répertoire de travail.
- Ajoutez une bétise dans `HelloWorld.java`. Ajoutez et commitez cette bétise (avec un message explicite).
- C'est une bétise. On veut revenir en arrière. Faites un `git log`, trouvez le hash de votre commit fautif puis `git revert le_hash_fautif`. Votre bétise est annulée.
- Nous allons maintenant regarder comment remiser une modification locale. Modifiez de nouveau `HelloWorld.java`. Faites un `git status`. Vous avez bien des modifications de votre répertoire de travail non prises en compte. Finalement, cette modification n'est peut être pas une bonne idée maintenant, mais vous voulez vous en souvenir. Vous allez la remiser. Pour cela, `git stash`. Un nouveau `git status` montre que ces modifications ont disparues. Elles sont toujours dans votre grenier/remise. Un `git stash pop` les ramène et un `git stash clear` vide votre grenier.
- `git push`

à faire à 2

- Allez dans gitlab sur votre projet, puis dans repository-> graph. Vous voyez un graphe qui est une chaîne.
- Placez-vous en binômes, par la suite les membres du binôme seront appelés Diabolo et Satanas, choisissez qui est qui dans votre binôme.
- Diabolo crée un groupe (depuis gitlab) sous l'oeil attentif de Satanas.
- Diabolo invite Satanas dans le groupe comme développeur.
- Diabolo va dans son projet et invite le groupe.
- Diabolo donne les droits "maintainer+developer" sur la branche principale (via le menu settings -> repository -> protected branch) afin que Satanas ait assez de droits.
- Satanas peut maintenant récupérer le projet avec un `git clone <l'adresse du projet>`.
- Effectuez ensuite les actions qui suivent.

Diabolo	Satanas
Modification de HelloWorld.java, add, commit	Modification de HelloWorld.java, add, commit

Diabolo	Satanas
push	
	push -> rejeté !
	pull
	si besoin : résolution des conflits en éditant le fichier fusionné
	add commit push
pull	
regarder le graphe (gitlab repository -> graph ou git log --graph)	regarder le graphe (gitlab repository -> graph ou git log --graph)
Modification de HelloWorld.java, add, commit	Modification de HelloWorld.java, add, commit
	push
fetch	
merge	
si besoin : résolution des conflits en éditant le fichier fusionné	
add commit push	
	pull
regarder le graphe (gitlab repository -> graph ou git log --graph)	regarder le graphe (gitlab repository -> graph ou git log --graph)

Ici Diabolo et Satanas travaillent tous sur la même branche, la branche principale. Ce n'est pas une très bonne pratique. Nous verrons plus loin comment travailler sur des branches séparées

Manipulations Maven de base (à réaliser seul, sans IDE)

Dans un terminal :

- Créez un nouveau projet Maven avec l'archetype quickstart :

  ```
  mvn archetype:generate -DgroupId=fr.umfds.agl -DartifactId=mon-projet
  -DarchetypeArtifactId=maven-archetype-quickstart -
  DinteractiveMode=false
  ```

  (voir cours)
- Regardez la structure induite pour votre projet et le POM obtenu
- Faîtes une première compilation :

  ```
  ```

```
mvn package
```

  - Que s'est-il passé ?
  - Selon la version de maven utilisée vous pourriez obtenir une erreur de compilation ... Dans ce cas, changez de version de Java pour une moins antique que celle par défaut (de préférence une qui est installée !) en ajoutant au POM quelque chose comme :

```
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>
```

- Faites un nettoyage :

```
mvn clean
```

  - Que s'est-il passé ?
- Ajoutez dans le fichier de test un test qui échoue (avec un assert(fail) par exemple)
- Recompilez
  - Que s'est-il passé ?
- Enlevez le test qui échoue
- Modifiez le fichier pom pour déclarer une dépendance vers une version la version 4.4 des Apache Commons Collections (dépôt maven)
  - Recherchez l'artefact dans le dépôt (ici)
  - Vous devriez arriver ici où vous trouverez la dépendance maven à coller dans le fichier pom.xml.
- Utilisez la dépendance :
  - Dans une nouvelle classe ou la classe existante, utilisez la classe ArrayListValuedHashMap<K,V> qui permet de manipuler des tableaux associatifs multi-valués (plusieurs valeurs par clef). Par exemple, vous pouvez modifier ainsi la classe App :

```
package fr.umfds.agl;// dépend de votre groupId
import
org.apache.commons.collections4.multimap.ArrayListValuedHashMap;
public class App
{
    public static void main( String[] args ){
    ArrayListValuedHashMap<String, Integer> map=new
ArrayListValuedHashMap<>();
    map.put("clef1", 1);
    map.put("clef1", 2);
    map.put("clef2", 3);

    System.out.println(map);
```

```
    }
}
```

- Compilez
  - que s'est-il passé ?
  - où sont les éléments téléchargés ?

Git et Maven sous IntelliJ

- Créez un nouveau projet maven :
  - File-> new Project
  - Choisir dans Generators (à gauche) : Maven Archetype
  - Donnez un nom au projet et placez-le où vous le souhaitez
  - Cochez Create Git repository
  - Sélectionnez l'archétype maven-archetype-quickstart
  - Dans advanced settings : donnez le GroupId de votre choix, et renommez si vous le souhaitez l'ArtifactId.
  - Puis cliquez sur Create.
- Quelques réglages du POM avant de commencer (en fonction de votre environnement et la version du quickstart archetype utilisée) :
  - ajustez la version de java source et cible

```
<properties>
    ...
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

  - ou mieux : utilisez une propriété pour la version !
  - mettez à jour le projet en cliquant sur un petit bouton avec le m de maven et un symbole de mise à jour bleu.
- lancez un build du projet :
  - dépliez la vue Maven (via le petit panneau à droite)
  - Dans le vue Maven dépliez le lifecycle du projet, et double-cliquez sur package
  - le superbe test de AppTest ne devrait pas détecter d'erreur et l'application devrait se construire sans erreur ...
- Placez ensuite l'ensemble des fichiers générés sur git
  - dépliez la vue Commit (via le petit panneau à gauche)
  - dans la vue Commit, sélectionnez tous les changements, mettez "initial commit" comme message de commit
  - cliquez sur commit and push

- le push a besoin de l'adresse du dépôt distant, et indique donc main -> Define remote.
- Cliquez alors sur Define remote
- Il vous est demandé l'adresse pour le dépôt distant.
- Allez sur gitlab.com, et créez un nouveau projet vide. Copiez l'adresse de ce projet.
- Collez l'adresse du projet comme remote url, pour origin.
- Validez, votre dépôt gitlab est maintenant le `origin` de votre projet
- Puis push.

Application à la gestion des TER

Lors de la séance précédente, nous avons commencé le développement de l'application de gestion des TER de M1. Le projet était un projet maven.

1. Placer le projet sur Git
    1. Ouvrez le projet dans IntelliJ.
    2. Dans le menu principal VCS, choisir `Enable Version Control Integration`.
    3. Sélectionner git et valider.
    4. Procéder ensuite comme précédemment pour commiter et placer les sources sur un dépôt distant.
2. Etudiez le fichier pom.xml : il explique pourquoi, sans avoir rien installé (ni Spring, ni ThymeLeaf notamment), votre projet a directement compilé : nous vous avions fourni le fichier indiquant toutes les dépendances nécessaires.
3. Etudiez le fichier `.gitignore` Il a été généré à partir du site dans le nom est donné en fin de fichier, bien pratique ...
4. Selon le temps qu'il vous reste, faîtes avancer votre projet.