

TD5 : Git et GitLab ISSUES, TASKS ET MILESTONES

Compétences

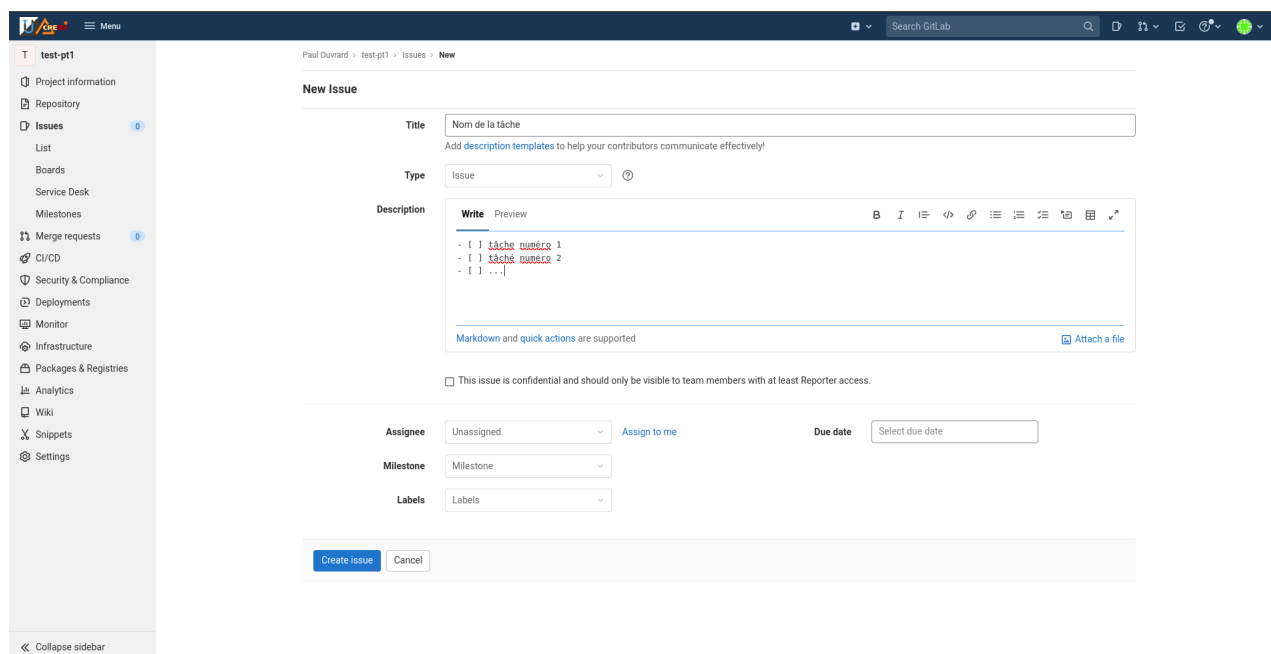
- Utiliser GitLab pour la gestion de projet
- Mettre en place les notions d'*issues* et de *tasks*
- Définir un jalon (*Milestone*) pour une release

GitLab et la gestion de projet

Issues et Tasks

Une *issue* est une fonctionnalité disponible sur GitLab (mais également GitHub) qui peut être considérée comme un besoin. Une issue peut très facilement être créée depuis le dépôt GitLab de votre projet à partir du menu (situé à gauche) Issue > New Issue. Vous pouvez alors donner un nom et une description à votre *issue*. Vous pouvez également l'assigner à un membre du groupe, préciser la date de rendu et y associer un jalon (*milestone*) dont nous parlerons juste après, ou des étiquettes. (*label*).

GitLab permet également d'associer à une issue une liste de tâches à réaliser. Vous pourrez alors cocher manuellement une des tâches de la liste dès que celle-ci aura été accomplie. Cela vous permettra d'indiquer à l'équipe le niveau d'avancement des différentes *issues*.



1. À partir des informations données ci-dessous, créez votre première issue correspondant au besoin : **exporter les données au format CSV**.

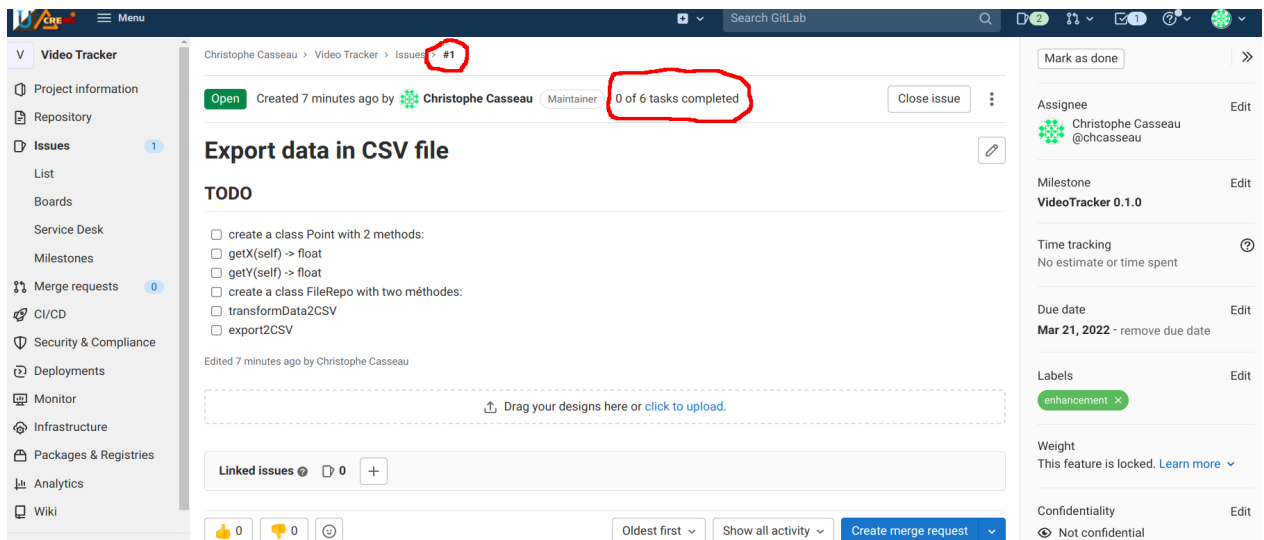
- **Issues** : <https://docs.gitlab.com/ee/user/project/issues/>

Il est possible de formater les commentaires de vos issues à l'aide d'un langage de balisage appelé **markdown** : <https://docs.framasoft.org/fr/grav/markdown.html>

- **Tasks** : You can use task lists to break the work for an issue or pull request into smaller tasks, then track the full set of work to completion.

To create a task list, preface list items with a hyphen and space followed by []. To mark a task as complete, use [x].

- **Labels** : Ils vous aident à organiser et étiqueter votre travail afin que vous puissiez suivre et trouver les éléments de travail qui vous intéressent. <https://docs.gitlab.com/ee/user/project/labels.html>



Dans cette image, nous pouvons voir une *issue* nommée "Export data in CSV file" contenant une liste des 6 tâches nécessaires pour réaliser l'issue. Nous pouvons noter qu'aucune de ces 6 tâches n'a pour le moment été accomplie (*0 of 6 tasks completed*). Enfin, nous pouvons observer que cette *issue* est la tâche numéro 1 (#1). Cette information est particulièrement intéressante puisque GitLab permet de modifier automatiquement l'état d'une tâche lors d'un commit. Pour cela, il faut indiquer dans le message du commit l'identifiant de la tâche en question ainsi que le sens du commit. Par exemple, `git commit -m "closes #1"` permet de fermer l'issue numéro 1. Pour connaître la liste des actions et des mots-clés disponibles, consultez la documentation de GitLab.

Milestone

Le jalon (*milestone* en anglais) est un point d'arrêt dans le processus permettant le suivi du projet. C'est l'occasion pour l'équipe de faire un bilan intermédiaire et de valider une étape avec des livrables comme des documents ou une version bêta du logiciel. L'objectif d'un jalon est de s'assurer à un moment donné :

- du bon déroulement de la phase précédente - notamment pour capitaliser sur ce qui a fonctionné ou pas et revoir potentiellement les pratiques pour la suite,
- de la validation des livrables attendus,
- de valider le lancement de l'étape suivante.

Là encore, GitLab permet de créer facilement des jalons à partir du menu Issues > Milestones > New milestone. Vous pourrez donner un nom à votre jalon, une description ainsi que des dates de début et de fin. Enfin, il vous sera possible de rattacher certaines issues à votre jalon :

- soit en éditant une issue préalablement créée et en lui affectant le jalon souhaité,
- soit directement depuis le jalon en cliquant sur New issue dans le menu de droite.

Semantic Versioning

Pour numéroter les versions de notre logiciel nous utiliserons la numérotation de type X.Y.Z dont la convention *Semantic Versioning* propose la signification suivante :

- **X** correspond à la version majeure : changements non rétrocompatibles. Les évolutions majeures apportent de nouvelles fonctionnalités, en changeant radicalement l'apparence ou l'architecture du logiciel,
- **Y** correspond à la version mineure : ajouts de fonctionnalités rétrocompatibles, principalement des corrections de bugs, ajouts de quelques fonctionnalités,
- **Z** correspond au correctif : corrections d'anomalies rétrocompatibles, failles de sécurité.

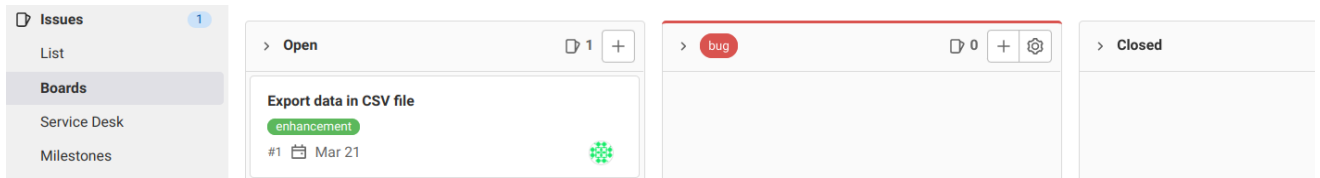
Cette convention est de plus en plus adoptée, notamment dans le milieu de l'open-source.

1. Créez un premier jalon pour le projet VideoTracker. Il devra comporter l'ensemble des besoins (issues) et tâches (tasks) nécessaires pour faire fonctionner la lecture de la vidéo uniquement image par image. Nous ne demandons aucune autre fonctionnalité liée à la lecture de la vidéo. Pour cela, vous devrez :

- planifier vos *issues*,
- attribuer à chaque issue les tâches correspondantes,

- associer à chaque *issue* et tâche un (ou plusieurs) membre(s) du projet.

2. Ajoutez ensuite dans le **issues boards** du projet une colonne **bug**, qui vous servira à repérer les dysfonctionnements de la version logiciel en cours. https://docs.gitlab.com/ee/user/project/issue_board.html



Travailler dans le terminal avec Git

Manipuler les fichiers de votre dépôt local

Ajout ou modification d'un ou plusieurs fichiers et commit...

```
git add file1 file2
git add file3
git commit -m "my message"
```

Suppression d'un fichier ou d'un répertoire

```
git rm file1
git rm -r dir1
git commit -m "my message"
```

Gérer les conflits

Les systèmes de contrôle de version sont, par essence, destinés à la gestion des contributions entre plusieurs auteurs distribués (généralement, des développeurs). Parfois, plusieurs développeurs peuvent essayer de modifier le même contenu plus ou moins en même temps. Cela peut alors engendrer ce que l'on appelle un *conflit*, qu'il vous faudra alors **impérativement** résoudre. Ce genre de problème peut survenir dans un scénario similaire à ceci :

1. le développeur A modifie la ligne 37 du fichier `toto.py`,
2. le développeur B décide de travailler en même temps, et édite lui aussi la ligne 37 du fichier `toto.py`,
3. une fois qu'il a terminé ses modifications, le développeur A *push* son travail sur le dépôt distant,
4. un peu plus tard, le développeur B souhaite à son tour synchroniser son travail sur le dépôt distant (`git push`).

Dans ce scénario précis, un conflit apparaît chez le développeur B, et sa demande de synchronisation avec le serveur distant échoue. En effet, les deux utilisateurs ont modifié quasi simultanément la même ligne du même fichier. Cependant, le développeur A ayant fait un `git push` en premier, il va alors modifier la version du fichier `toto.py` du dépôt distant. Lorsque le développeur B va demander à son tour la synchronisation, Git va considérer que celui-ci n'est pas à jour (son dépôt local est en retard d'au moins un commit par rapport au dépôt distant) et va alors refuser cette synchronisation. Pour pallier ce problème, le développeur B n'a pas d'autre choix que de mettre à jour son dépôt local à l'aide de `git pull`. Cela va avoir pour effet de rapatrier les modifications effectuées par le développeur A sur son dépôt local. Le fichier `toto.py` va alors contenir quelque chose comme ceci autour de la ligne 37 :

```
<<<<<< HEAD
version du développeur B
=====
l'autre version qui vient du dépôt distant (modifications faites par le développeur A)
>>>>>> <commit>
```

Pour résoudre le conflit, il suffit au développeur B d'éditer à la main ce texte en choisissant la version qu'il juge la plus appropriée (et en effaçant tout le reste) à l'aide d'un éditeur de texte ou d'un IDE (comme Visual Studio Code) avant de faire de nouveau un `git push`.

Cet exemple illustre tout particulièrement l'importance de synchroniser au maximum votre dépôt local avec le dépôt distant. Plus les différences entre les deux sont importantes, et plus vous avez de risque de créer des conflits. Notons

cependant que la commande `git fetch` permet de savoir si votre dépôt local est en retard d'un ou plusieurs commits par rapport au dépôt distant, sans rapatrier les modifications présentes sur le dépôt distant le cas échéant. Cette commande peut donc s'avérer particulièrement utile avant de faire un `git push` pour savoir si celui-ci ne risque pas engendrer de conflit.

Enfin, notons que si les modifications apportées par les développeurs A et B ne portent pas sur la même ligne ou concernent des fichiers différents, alors le `git pull` réalisé par le développeur B ne nécessitera pas d'aller éditer le fichier en question comme précédemment. Le développeur B n'aura qu'à faire un nouveau `git push` pour envoyer ses modifications sur le dépôt distant.

Exemple

Dans cet exemple, nous allons simuler le développeur B en modifiant directement le fichier `README.md` sur le dépôt GitLab distant. Le développeur A, donc vous, devra modifier ce même fichier en local sur son ordinateur.

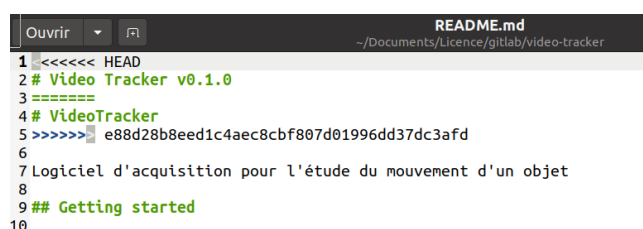
1. Éditez le fichier `README.md` à partir de la fenêtre graphique de votre dépôt distant GitLab.
2. Modifiez le titre de `README.md`, puis effectuez un commit en cliquant sur le bouton qui se trouve en bas de page.
3. Ouvrez ensuite le fichier `README.md` de votre dépôt local (c'est-à-dire celui qui se trouve sur votre ordinateur).
4. Modifiez également le titre de telle manière que le contenu soit différent de celui qui se trouve sur le dépôt distant.
5. Dans votre terminal, effectuez un commit puis un push. Normalement vous devriez avoir une erreur du type : **rejected**

```
^C(base) casseau@casseau-laptop:~/Documents/Licence/gitlab/video-tracker$ git co
it -am "modif_title"
[main c496d11] modif_title
1 file changed, 1 insertion(+), 1 deletion(-)
(base) casseau@casseau-laptop:~/Documents/Licence/gitlab/video-tracker$ git push
To gitlab.emi.u-bordeaux.fr: /video-tracker.git
! [rejected]        main -> main (fetch first)
error: impossible de pousser des références vers 'git@gitlab.emi.u-bordeaux.fr:c
hcasseau/video-tracker.git'
astuce: Les mises à jour ont été rejetées car la branche distante contient du tr
avail que
astuce: vous n'avez pas en local. Ceci est généralement causé par un autre dépôt
poussé
astuce: vers la même référence. Vous pourriez intégrer d'abord les changements d
istants
astuce: (par exemple 'git pull ...') avant de pousser à nouveau.
astuce: Voir la 'Note à propos des avancées rapides' dans 'git push --help' pour
plus d'information.
(base) casseau@casseau-laptop:~/Documents/Licence/gitlab/video-tracker$
```

6. Git vous signale que "la branche distante contient du travail que vous n'avez pas en local", autrement dit que votre dépôt local est en retard d'au moins un commit par rapport au dépôt distant (GitLab). Il vous conseille alors de faire un *pull* pour mettre à jour votre dépôt local.

```
(base) casseau@casseau-laptop:~/Documents/Licence/gitlab/video-tracker$ git pull
remote: Enumerating objects: 36, done.
remote: Counting objects: 100% (36/36), done.
remote: Compressing objects: 100% (29/29), done.
remote: Total 33 (delta 9), reused 15 (delta 4), pack-reused 0
Dépaquetage des objets: 100% (33/33), 3.73 Kio | 224.00 Kio/s, fait.
Depuis gitlab.emi.u-bordeaux.fr: /video-tracker
da074df..e88d28b main -> origin/main
Fusion automatique de README.md
CONFLIT (contenu) : Conflit de fusion dans README.md
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
(base) casseau@casseau-laptop:~/Documents/Licence/gitlab/video-tracker$
```

7. Après avoir fait le `git pull`, Git vous indique un **conflit de fusion**. Ouvrez le fichier `README.md` de votre dépôt local puis faites le nécessaire pour régler le conflit.



```
Ouvrir  README.md
~/Documents/Licence/gitlab/video-tracker
1 <<<<< HEAD
2 # Video Tracker v0.1.0
3 =====
4 # VideoTracker
5 >>>>> e88d28b8eed1c4aec8cbf807d01996dd37dc3afd
6
7 Logiciel d'acquisition pour l'étude du mouvement d'un objet
8
9 ## Getting started
10
```

8. Terminez avec un *commit* puis un *push* afin de synchroniser votre dépôt local avec le dépôt GitLab.