Git

Julien Romero[1]

November 13, 2017

[1]Télécom ParisTech, C201-6

Le contrôle des versions

Un contrôleur de version : pour quoi faire ?

- Retourner à un ancien état (avant l'apparition d'un bug par exemple)
- Voir les modifications chronologiquement
- Savoir qui a fait quelle modification
- Récupérer des fichiers perdus
- Organiser un projet
- ...

Git

Git n'enregistre que les modifications sur les fichiers.

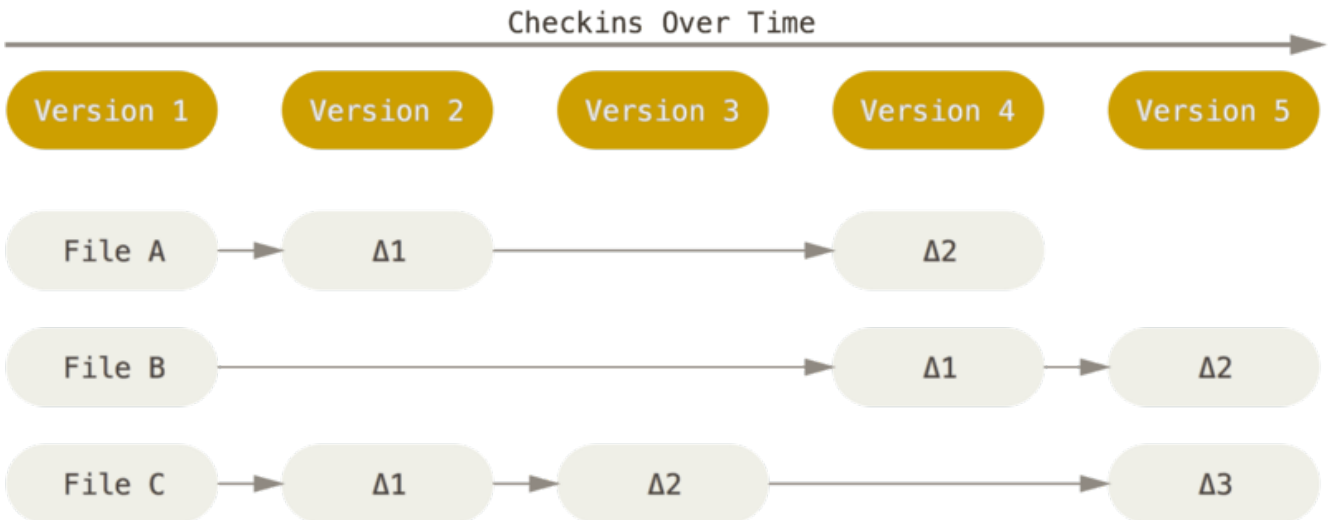


Figure 1: Les versions sous Git, tiré de https://git-scm.com

Git : les états d'un fichier

Sous Git, un fichier peut être dans des états différents:

Commited toutes les modifications faites précédemment au fichier sont enregistrées avec Git.

Modified le fichier a été modifié (par exemple, vous êtes en train de coder dedans) et les modifications n'ont pas encore été enregistrées sous Git

Staged on a signalé à Git que le fichier va faire partie de la prochaine version

Configurer Git

Configurer son ordinateur

Vous devez installer Git (il est déjà installé sur les machines de l'école). Vous pouvez ensuite modifier votre nom, email et l'éditeur de texte par défaut de Git.

Configuration (en ligne de commande)

git config - -global user.name "John Doe"
git config - -global user.email johndoe@example.com
git config - -global core.editor emacs

Générer une clef SSH

Une clef SSH vous permet de vous identifier facilement sur le dépot. Elle se compose d'une clef publique (que vous pouvez partager) et d'une clef privée (qui **doit** rester secrète).

Générer une clef SSH (en ligne de commande)

```
ssh-keygen -t rsa -b 4096 -C "email@example.com"
ssh-add ~/.ssh/id_rsa
```

Enregistrer sa clef SSH

Dans Gitlab, allez dans "Settings" (en haut à droite) puis dans "SSH Keys". Dans la partie "Key", copiez le contenu du fichier ~/.ssh/id_rsa.pub (attention, on partage seulement la clef publique). Vous pouvez ajouter un titre à votre clef. Appuyez sur "Add Key". Vous pouvez répéter ces opérations pour chaque nouvel ordinateur.

Cloner votre dépot

Pour récupérer le contenu de votre projet et initialiser le dossier git, entrez la commande:

Cloner un dépot

git clone git@gitlab.enst.fr:pact/2017-2018/pact4X.git

Un nouveau dossier est créé et il est prêt à être utilisé. Normalement, cette commande est utile une seule fois par ordinateur.

Les commandes de base

Voir les dernières versions

On peut accéder aux dernières versions de notre application en tapant:

Afficher dernières versions
git log

On a accès à:

- Un numéro unique par commit
- L'auteur de la version
- La date de la modification
- La description de la nouvelle version

On peut aussi utiliser **tig** pour avoir un accès plus visuel.

Récupérer les modifications sur le dépot

Les fichiers sur le dépot et ceux sur votre ordinateur peuvent différer si quelqu'un soumet une nouvelle version. Pour récupérer la version la plus récente :

Récupérer le dépot

git pull

Vérifier l'état local

État de Git

git status

On apprend :

- Notre position par rapport au dépot commun (en avance, synchronisé,...)
- Les fichiers modifiés
- Les fichiers non suivis, pas du tout connus par Git

Voir les modifications sur un fichier

Avant de soumettre les modifications, vous pouvez visualiser les modifications faites à un fichier à tapant:

Voir modifications fichier

git diff nom_du_fichier

Ajouter des modifications

Pour que Git considère les modifications faites à un fichier dans la prochaine version, tapez:

Ajouter des modifications
git add nom_des_fichiers

Sur les nouveaux fichiers, ceux-ci seront rajouter à Git.
Attention Les commandes "git add *" et "git add ." sont à proscrire, elles peuvent être très dangereuses (c.f. plus tard).

Retirer et déplacer des fichiers

Ces commandes sont très similaires aux commandes Unix.

Effacer un fichier

git rm nom_fichier

Déplacer un fichier

git mv nom_fichier

Enregistrer les changements dans une nouvelle version

Une fois que les fichiers modifiés sont ajoutés à la prochaine version, celle-ci peut être créée.

Soumettre une nouvelle version

git commit

Un éditeur de texte s'ouvre afin d'écrire le message du commit. On peut écrire le message directement en ligne de commande.

Soumettre une nouvelle version directement avec un message

git commit -m "votre message"

On peut aussi s'affranchir de "git add" pour les fichiers modifiés mais non nouveaux.

Soumettre dans une nouvelle version tous les fichiers modifiés

git commit -am "votre message"

Les messages de commit

Les messages de commit sont **importants**. Ils servent à :

- Comprendre les modifications faites
- Savoir pourquoi les modifications ont été faites
- Structurer le projet
- Revenir à des versions précises
- Documenter

Les règles d'un message de commit

- Séparer le sujet du commit de la description par une ligne vide (pas compatible avec git commit -m). Utile avec git log –oneline
- Le sujet doit être court (moins de 50 caractères)
- Commencer le sujet par une majuscule
- Pas de point à la fin du sujet
- Utiliser l'impératif pour le sujet (ex: "Add new windows to the website")
- Limiter les lignes à 72 caractères dans la description
- Dans la description, expliquer *quoi*, *pourquoi* et *comment*

(tiré de https://chris.beams.io/posts/git-commit/)

Envoyer ses versions au dépot distant

Une fois qu'une ou plusieurs version sont créées localement, on peut les envoyer sur le dépot commun.

Envoyer les nouvelles versions

git push

À noter qu'il faut d'abord avoir fait un *git pull*.

Les branches

Les branches vont permettre de développer de nouvelles fonctionnalités en parallèle sans que le travail des autres n'influe sur le notre.

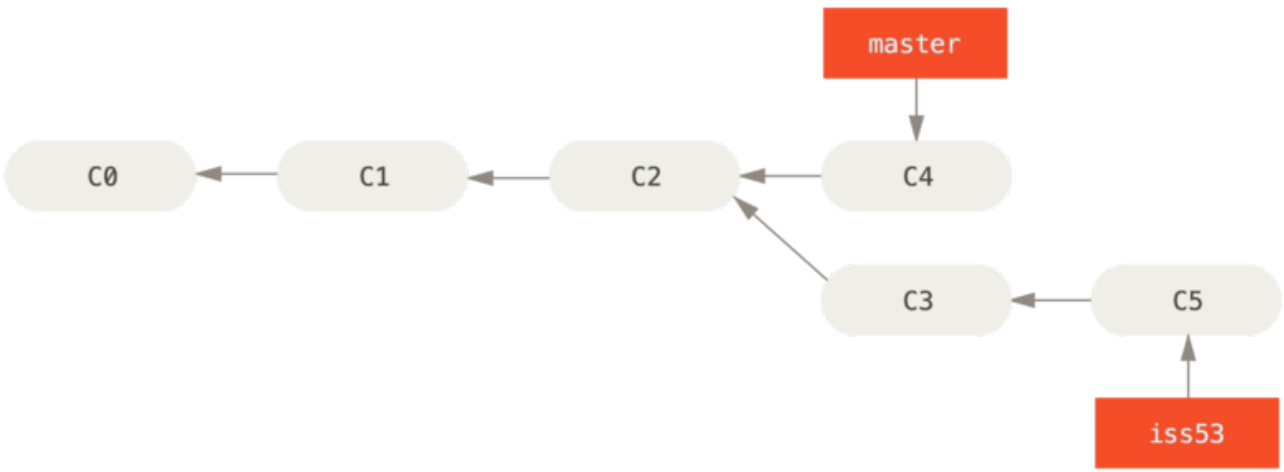


Figure 2: Les branches sous Git, tiré de https://git-scm.com

Créer des branches

Lorsqu'une nouvelle branches est créée à partir d'une autre, les versions entre ces deux branches vont être indépendantes et auront seulement en commun un ancêtre.

Créer une nouvelle branche

git checkout -b nouvelle_branche

Se déplacer entre les branches

Les déplacements

git checkout nom_branche

git checkout numero_commit

git checkout master

Attention, *git checkout fichier* retire tous les changements faits à un fichier depuis le dernier commit.

Fusionner des branches

Tout d'abord, allez sur la branche qui va accueillir la fusion, par exemple pour master *git checkout master*. Ensuite, vous pouvez fusionner l'autre branche

Fusionner deux branches
git merge autre_branche

Si tout se passe bien, la fusion est automatique. On peut ensuite supprimer l'ancienne branche.

Supprimer une branche
git branch -d autre_branche

Fusion non-automatique

Si la fusion ne peut pas se faire de manière automatique, il vous faut choisir quelle version garder dans les fichiers à problèmes. Vérifiez quels sont ces fichiers avec *git status* et modifiez les avec votre éditeur de texte. Un conflit se présente comme ceci:

Conflit

«««< HEAD:index.html
contact : email.support@github.com
=======

please contact us at support@github.com

»»»> prob53:index.html

Supprimez ce qui est inutile, et commitez.

GitLab en pratique

Issues

Les *issues* permettent de commencer les discussions. On peut les utiliser pour:

- Discuter des idées
- Donner des idées de fonctionnalités
- Poser des questions
- Reporter des bugs
- Obtenir de l'aide/du support
- Discuter une nouvelle implémentation

Les issues sont créées directement dans GitLab.

Coder une issue

Dans une *issue*, on peut désigner un responsable, qui devra implémenter la fonctionnalité. Ce responsable peut changer au cours du temps, en particulier quand plusieurs personnes sont impliquées dans le développement. On peut aussi ajouter une date limite, des points repères (Milestone) et des Labels.
Pour montrer que l'on commence à coder une *issue*, on ajoutera au premier message de commit "Ref #xxx" où xxx est le numéro de l'*issue*. En général, une nouvelle branche sera créée pour l'issue.

Merge Requests

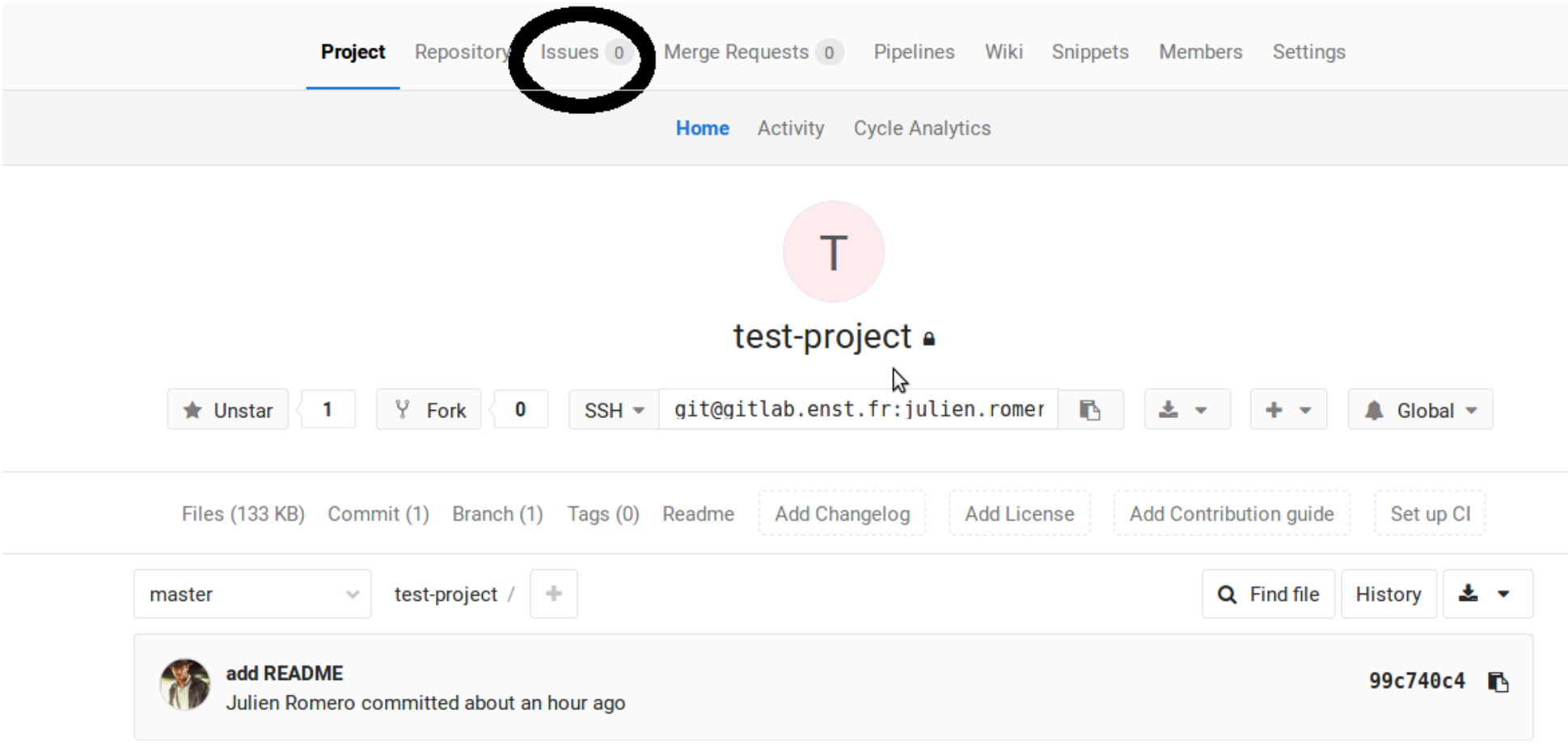
Les *Merge Requests* vont faciter l'incorporation des nouvelles fonctionnalités et permettent de fermer certaines *issues*. A la fin, une *Merge Request* agira comme un merge. Il est aussi possible de faire des retours sur le code avant de faire le merge.
Les *Merge Requests* sont créées directement dans GitLab.
Pour indiquer qu'une *Merge Request* n'est pas prête à être résolue, on ajoutera "WIP:" devant le titre.

Workflow

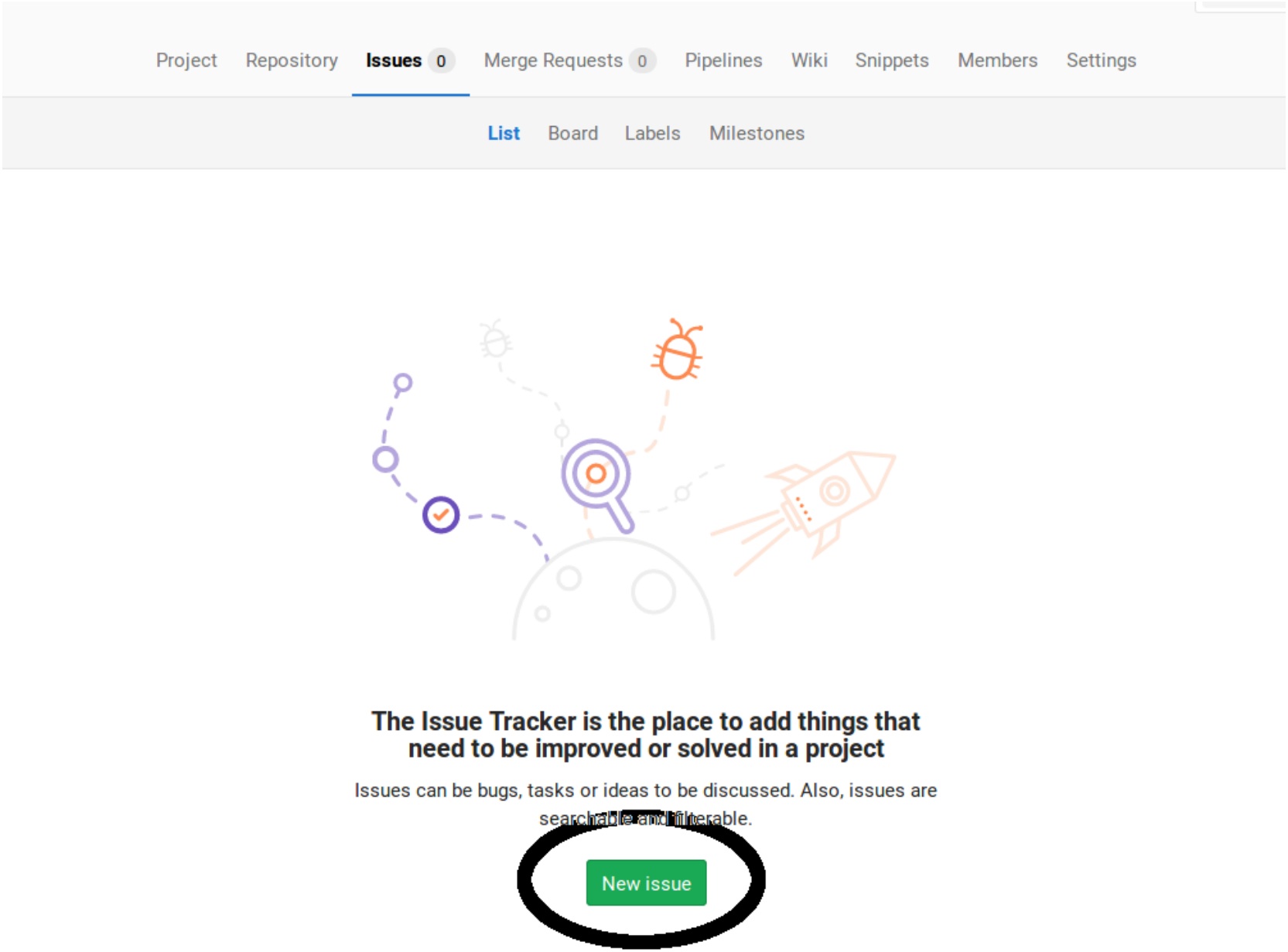
Pour implémenter une nouvelle fonctionnalité, on suivra ces étapes:

1. Créer une *issue*.
2. Attribuer un responsable.
3. Créer une nouvelle branche pour l'implémentation. On n'oubliera pas de référencier l'*issue*.
4. Créer la *Merge Request* avec l'indication "WIP:".
5. Implémenter la fonctionnalité. Si nécessaire, on changera le responsable.
6. Une fois la fonctionnalité créée, on retire le "WIP:" sur la *Merge Request* et on demande des retours.
7. Quand tout le monde est d'accord sur la *Merge Request*, on l'accepte et la fusion se fait.

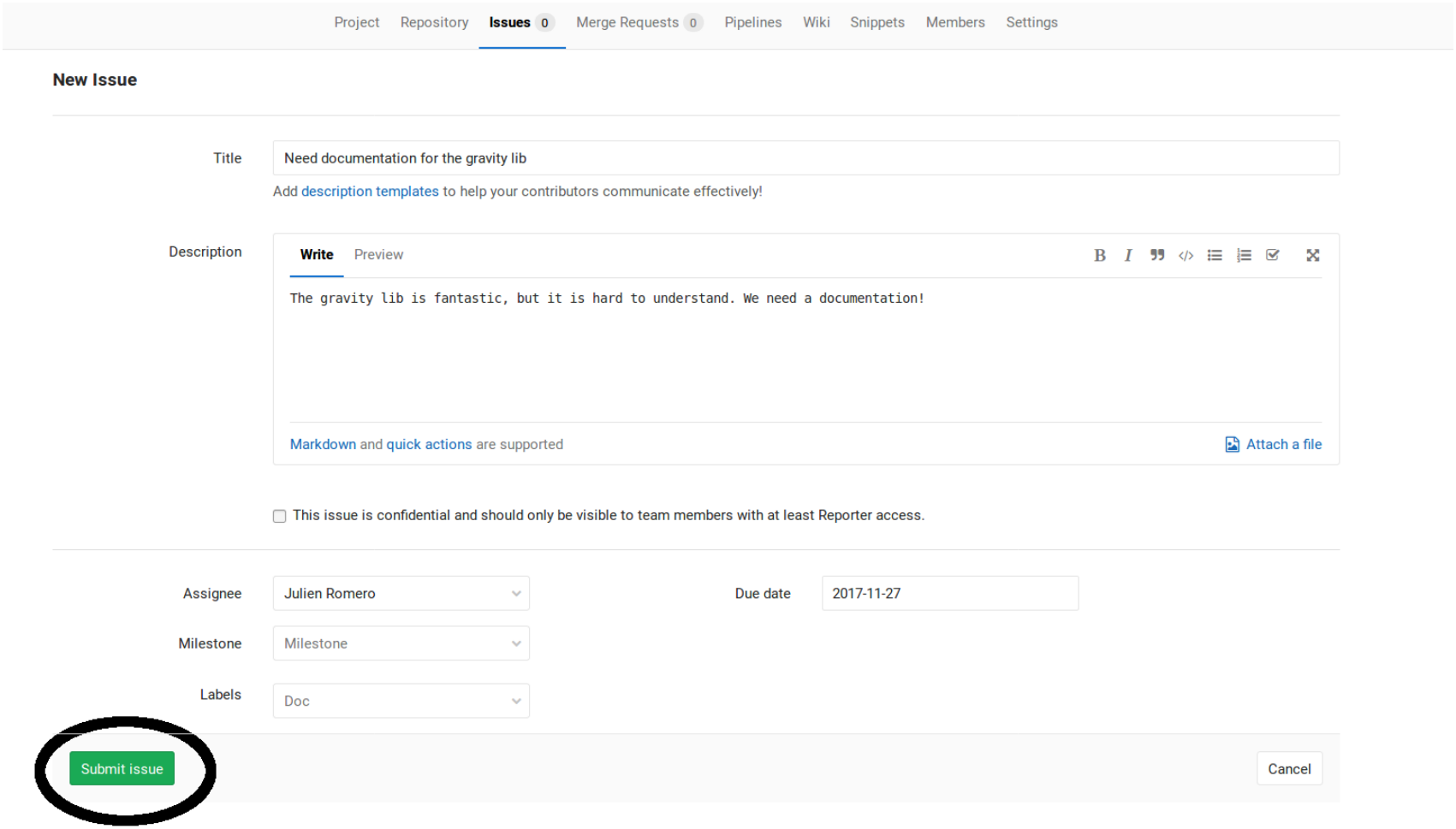
Création de l'issue



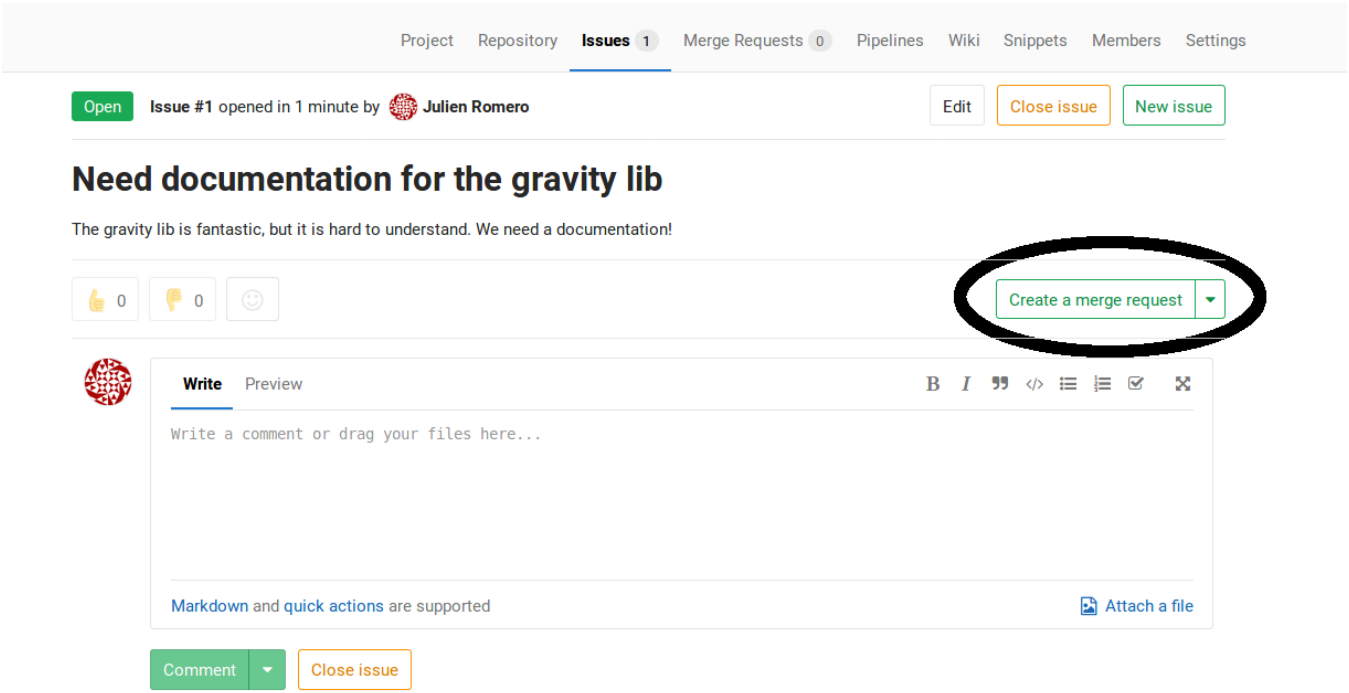
Création de l'issue



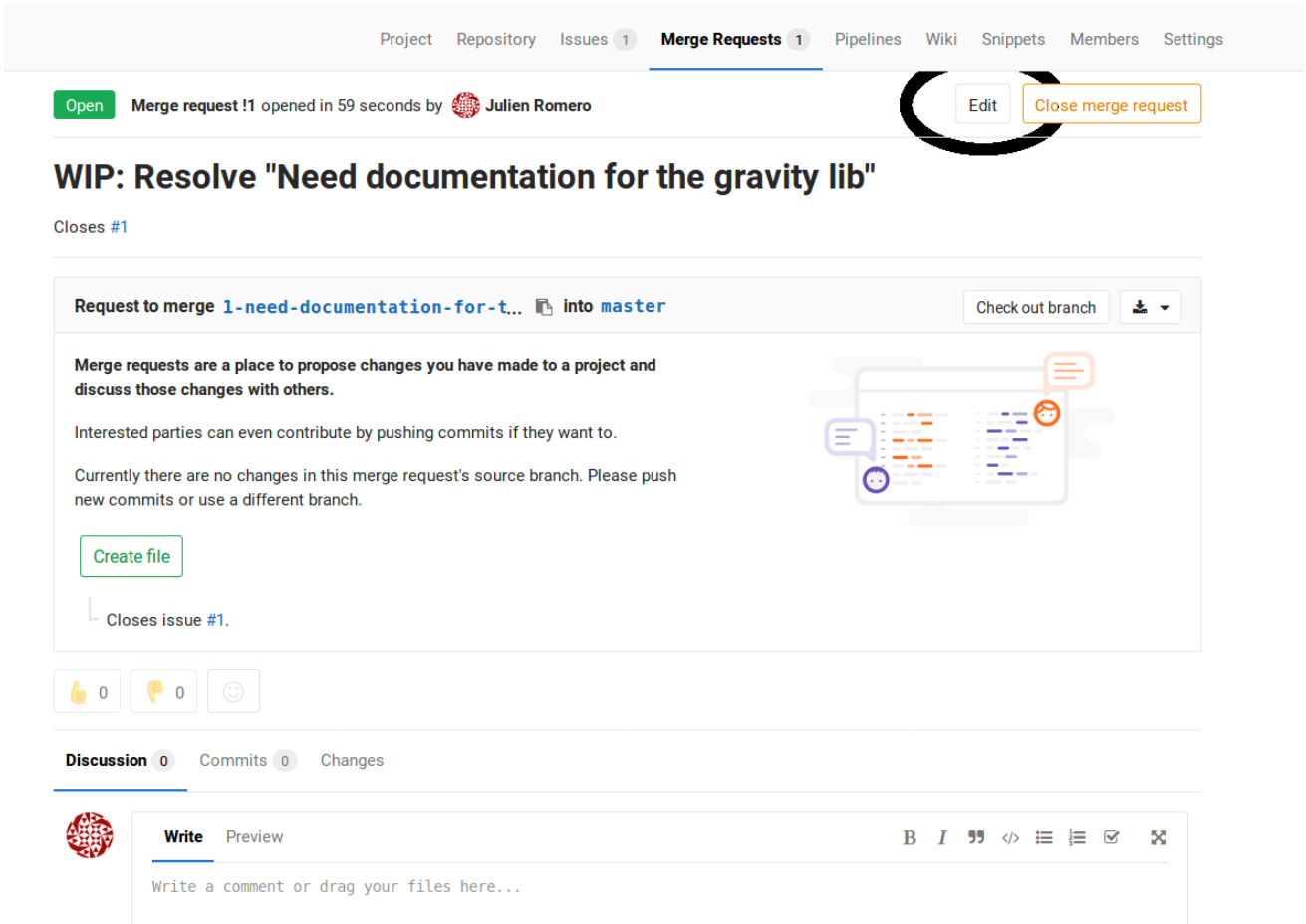
Création de l'issue



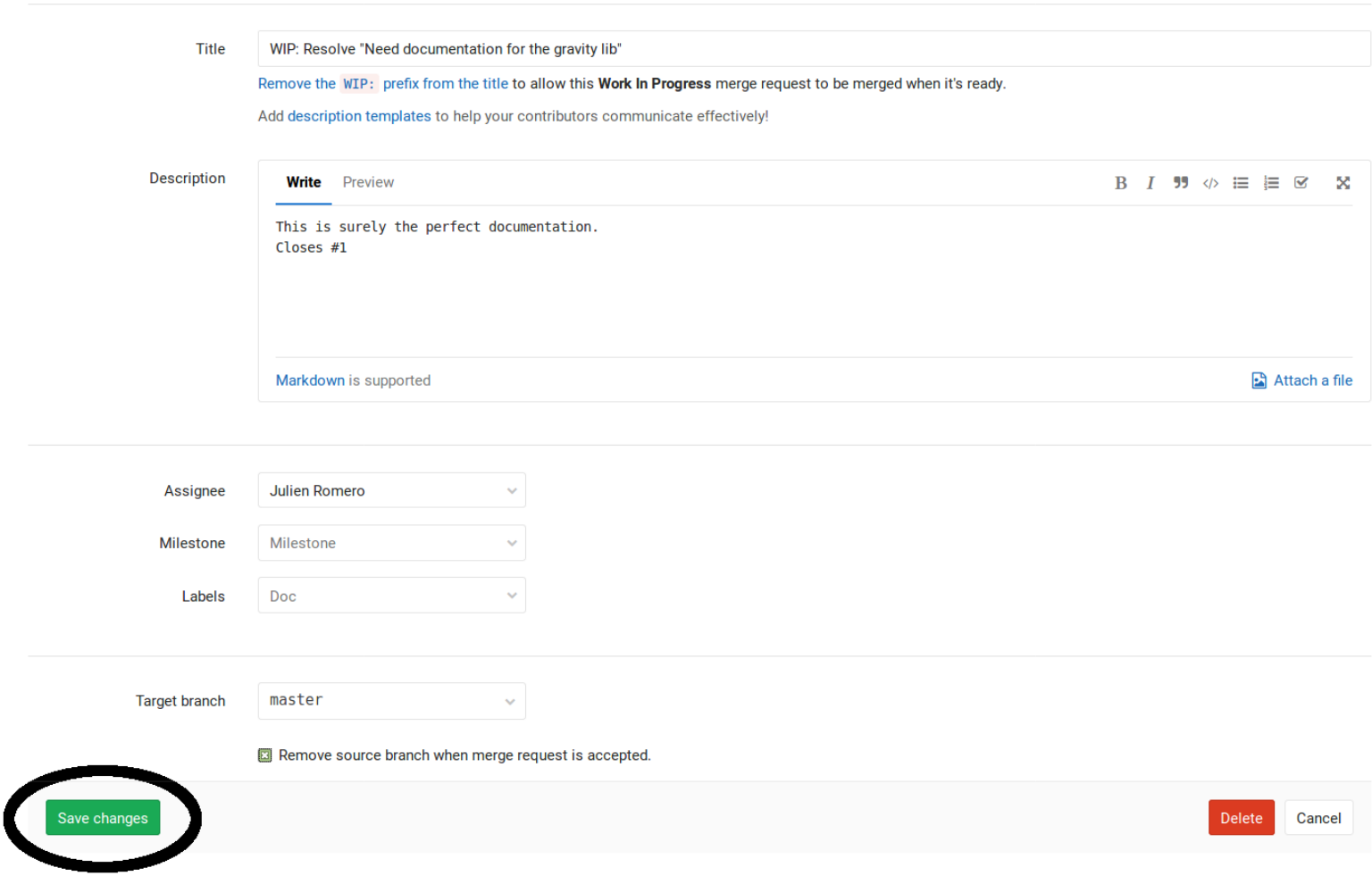
Création du Merge Request



Création du Merge Request



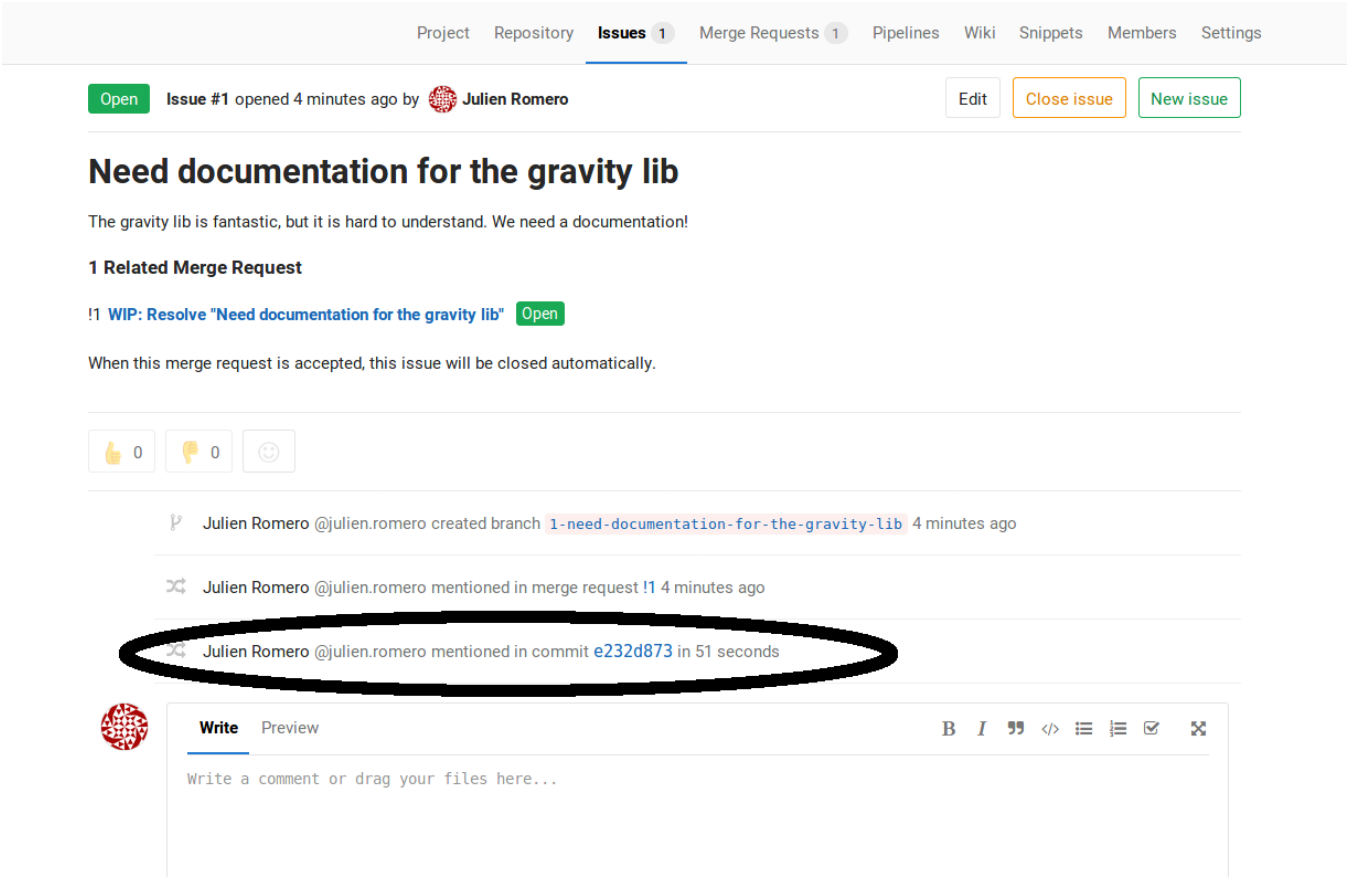
Création du Merge Request



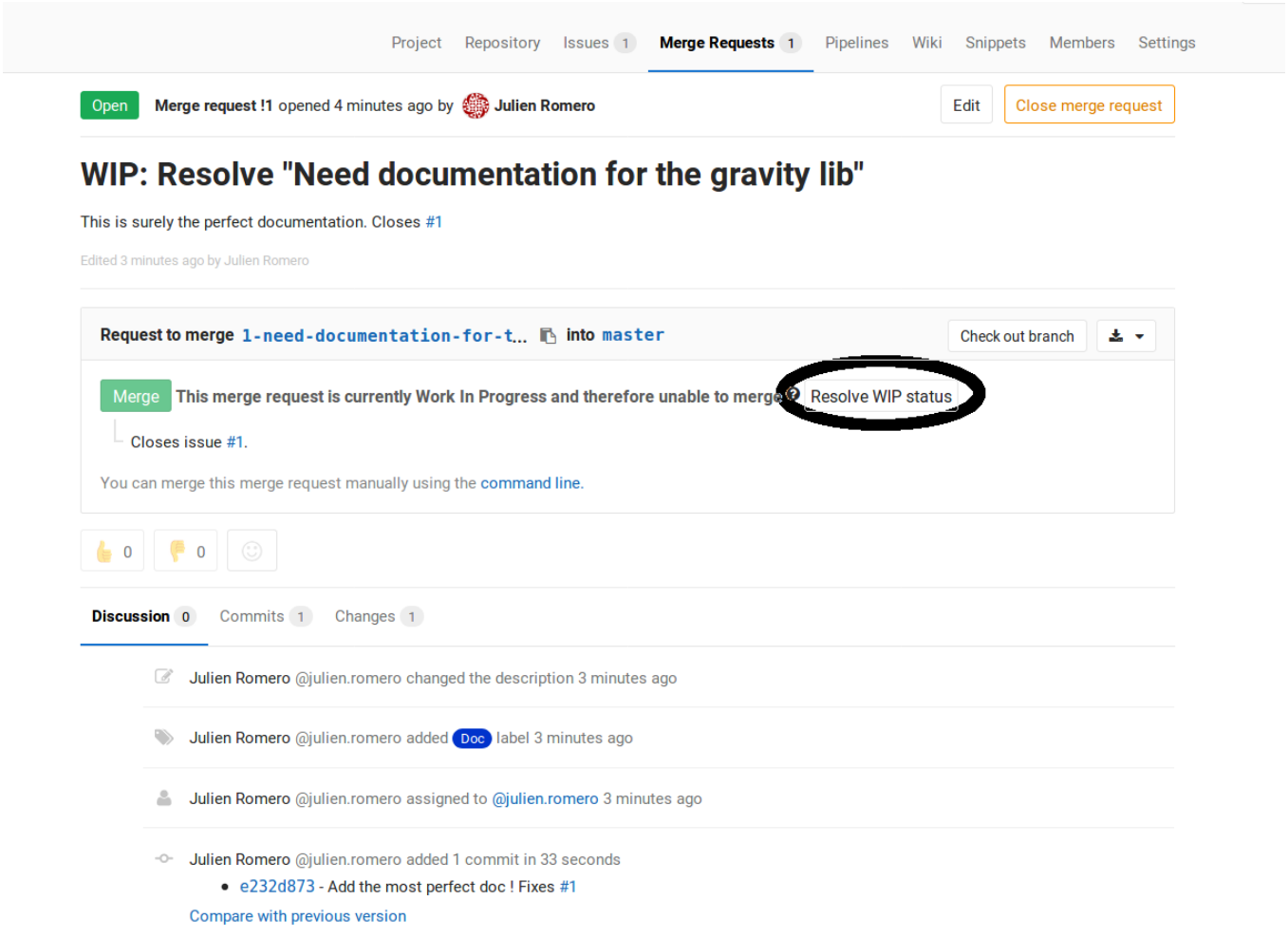
Implémentation

Dans la branche qui sera créée par GitLab, vous implémentez la résolution de l'issue. Vous n'oublierez pas de référencer l'issue pour pouvoir accéder aux commits plus facilement depuis l'issue.
On doit aussi discuter dans le Merge Request des modifications faites et de comment améliorer le code si nécessaire. Ces discussions peuvent se faire directement dans le code, dans la catégorie *changes*.

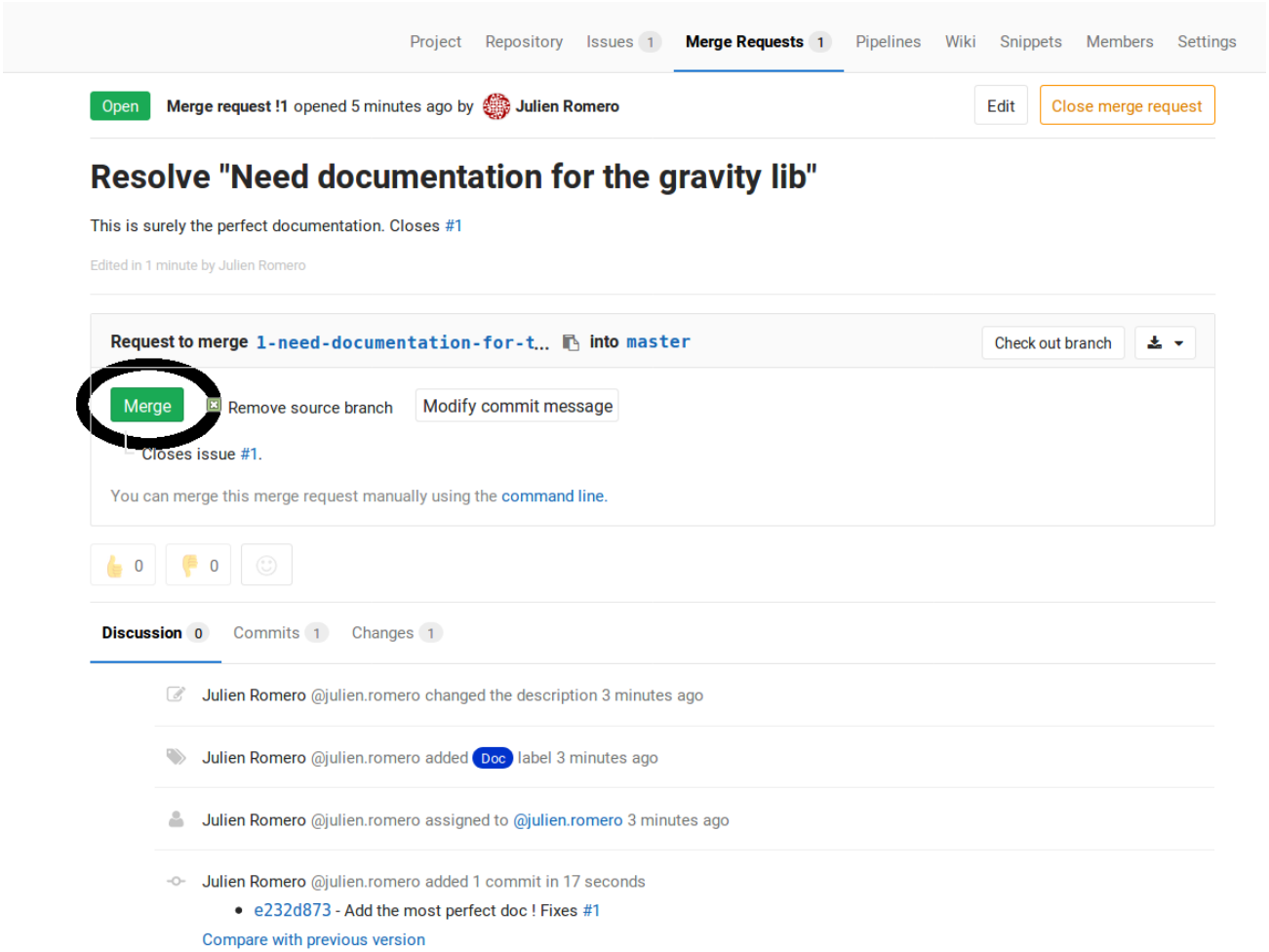
Référence dans l'issue



Quitter le status WIP



Fusionner les branches



Conseils pratiques

Que faut-il commit?

On commit:

- Les fichiers texte
- Le code source
- La doc
- Les fichiers de configurations

On évite de commit (trop souvent):

- Images
- PDF
- Plus généralement, tous les fichiers volumineux

On ne commit pas:

- Les fichiers compilés ou binaires (ex: *.class)
- Les fichiers créés par l'éditeur de texte (ex: *.swp)
- Une API/bibliothèque externe

.gitignore

Le fichier .gitignore permet d'ignorer les fichiers dans un certain format. On mettra par example *.class, *.swp, ... dedans.
Renseignez-vous sur des .gitignore déjà existants (par exemple https://gist.github.com/iainconnor/8605514 pour Android Studio) pour ne pas salir votre dépot de fichiers inutiles.

La branche master

La branche master est la branche principale de votre projet, sur laquelle on arrive quand on clone votre dépot. Dans l'idéal, elle contiendra constamment un prototype (même très léger) qui compile et fonctionne. Pour garder cette branche propre, on créera des branches auxiliaires.

References

- Livre officiel et gratuit sur Git
- Documentation de GitLab
- Le workflow dans GitLab
- Écrire un message de commit

To Do List

- S’enregistrer sur le gitlab de PACT si ce n’est pas déjà fait et m’envoyer un mail.
- Créer des Milestones pour les PANs et pour certains modules si nécessaire (voir avec l’expert).
- Créer des labels pour chaque modules et pour certaines actions (c.f. la doc pour des idées).
- Compléter ce que vous pouvez dans la partie rapport (quelques fichiers ont déjà été créés). Vous découperez en plusieurs issues pour que tout le monde utilise Git et le workflow (au moins un commit par personne).
- Commencer à créer des issues pour certaines fonctionnalités.

To Do List 2

Créez une structure de dossiers standard:

- Un dossier par programme séparé (client, serveur, android, programme en C...)
- Pour un programme en Java pour PC, utilisez la structure:
  src/org/pact/pactXY/moduleZ pour chacun de vos modules
  test/ pour les fichiers de test
  (bin/ qui est dans .gitignore)
  .classpath (ou l'équivalent IntelliJ)
  .project
- Pour un programme Android, utilisez la structure générée par Android Studio. S'il y a des sous-modules, les mettre dans app/src/main/java/org/pactXY...