

Git continued: contributing

Michaël Van Canneyt

September 27, 2021

Abstract

In a previous article, we introduced the distributed version system `Git`, and we showed how to use it to fetch and update sources from a remote repository. In this article, we'll show you how to save changes you made and how to send these changes to the remote repository.

1 Introduction

Using `Git` just to download some files from a project that you want to use is somewhat overkill: The whole point of using `git` is to be able to make changes, and to send those changes back to the remote repository from where you got the sources.

If you do not plan to contribute to a project, it is much easier to just download a zip with the sources. Both `Gitlab` and `Github` automatically make an URL available to download the sources of a branch of a project. for `Gitlab`, this URL is:

```
https://gitlab.com/PROJECT/-/archive/BRANCH/REPONAME-BRANCH.zip
```

Here you must replace `PROJECT` with the URL of the project repository, `BRANCH` with the branch name and `REPONAME` with the name of the repository.

For example, to download a zip with the latest sources of `FPC`, this is the URL to use:

```
https://gitlab.com/freepascal.org/fpc/source/-/archive/main/source-main.zip
```

The extension can be changed to other formats such as `.tar`, `.tar.gz` or `tar.bz2`, if you prefer one of these formats: `Gitlab` and `Github` will determine the archive format to use from the extension you added.

A similar mechanism exists for `github`:

```
https://github.com/PROJECT/archive/refs/heads/BRANCH.zip
```

So for the Free Pascal sources, the URL becomes:

```
https://github.com/fpc/FPCSource/archive/refs/heads/main.zip
```

But a source code management system such as `Git` is meant to be able to change the sources, and send the changes back to the originating repository:

If you have downloaded the sources of some project and you wish to contribute some changes back to the project, 2 steps must be done:

1. Record the changes locally (called committing).

2. Send the changes back to the remote repository (called pushing)

This is different from version systems like Subversion, where a commit is immediately sent to the remote repository.

We'll take a look at both steps in more detail.

2 Recording a change: what must be recorded ?

Before recording changes, you need to know what the changes are that need to be recorded. Git can tell you what files you have changed (for the files it is tracking) and what files are in your copy of the repository that it does not know about.

The Git command which tells you this, is called `status`.

On the command-line, it looks like this:

```
> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   myunit.pp

Untracked files:
  (use "git add <file>..." to include in what will be committed)
myotherunit.pp

no changes added to commit (use "git add" and/or "git commit -a")
```

How to interpret this?

- The list of files after `Changes not staged for commit` is the list of files that are changed locally.
- The list of files after `Untracked files` is a list of files that exist in your copy of the sources, but which Git is not tracking.

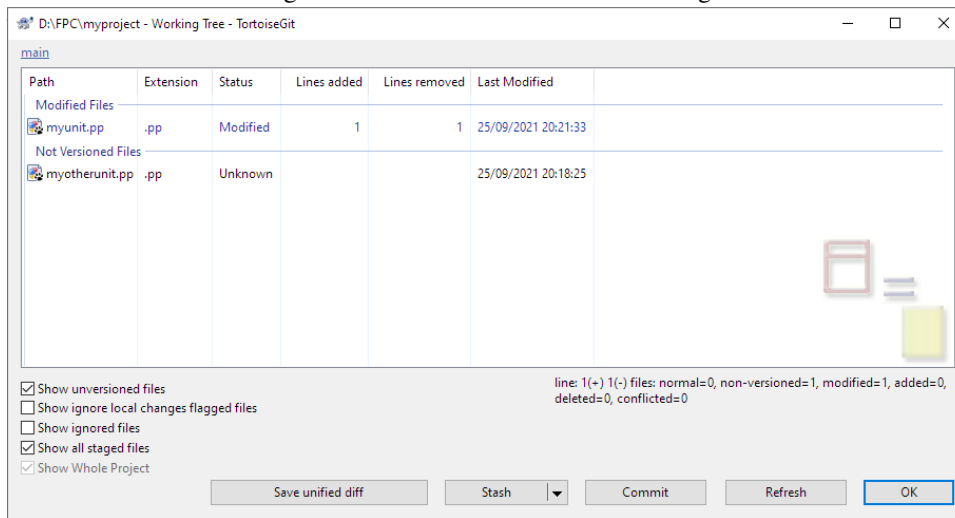
If you work with TortoiseGit, then you can see the status of the files right in the Windows Explorer, because the file icons will have a marker if they are changed:

D:\FPC\myproject

Name	Date modified	Type	Size
.git	25/09/2021 20:21	File folder	
myotherunit.pp	25/09/2021 20:18	Pascal Source Code	1 KB
myunit.pp	25/09/2021 20:21	Pascal Source Code	1 KB

Unversioned files have no marker. For more detail, you can use the context menu of the explorer to select `Git check for modification`. Doing so will pop up the dialog shown in figure 1 on page 3, which shows essentially the same information as the `git status` command. You can modify the view by checking or unchecking some of the options in the lower-left corner.

Figure 1: TortoiseGit modifications dialog



3 Recording a change: Adding a new file

The simplest case of a change you can make is simply adding a new file to the repository. Git needs to be told that you wish to start tracking changes to a file. So, the file must be added the repository.

Adding a file is done – not surprisingly – with the `add` command, for example:

```
git add myotherunit.pp
```

After this, Git knows you want to track changes to the file `myotherunit.pp`.

You can check this using the `status` command again:

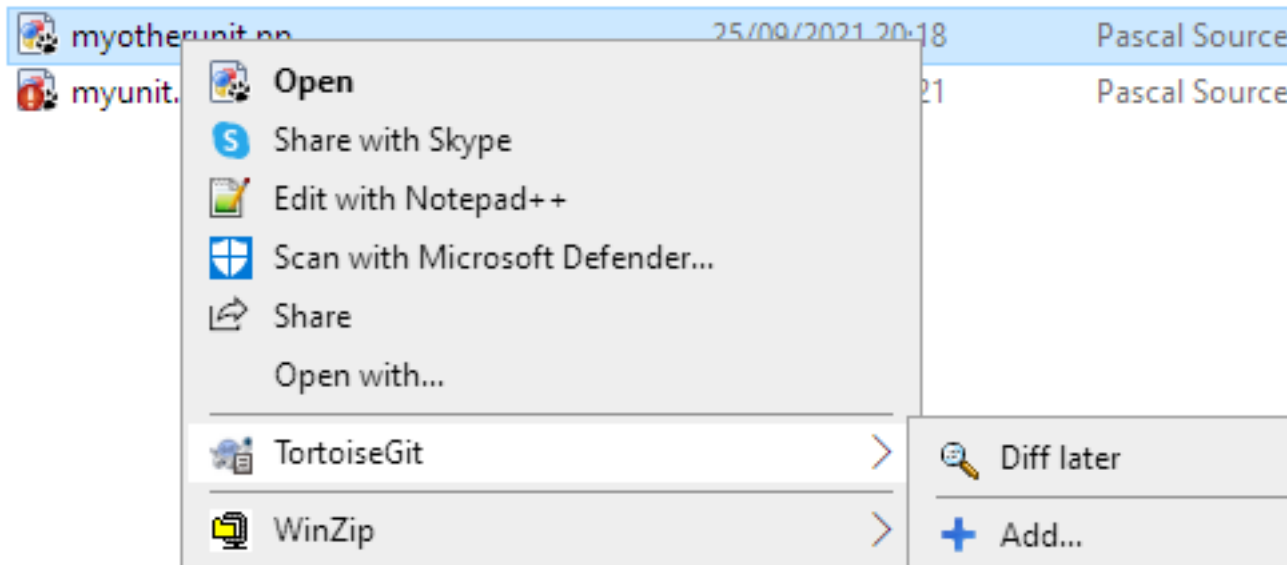
```
> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
new file:   myotherunit.pp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   myunit.pp
```

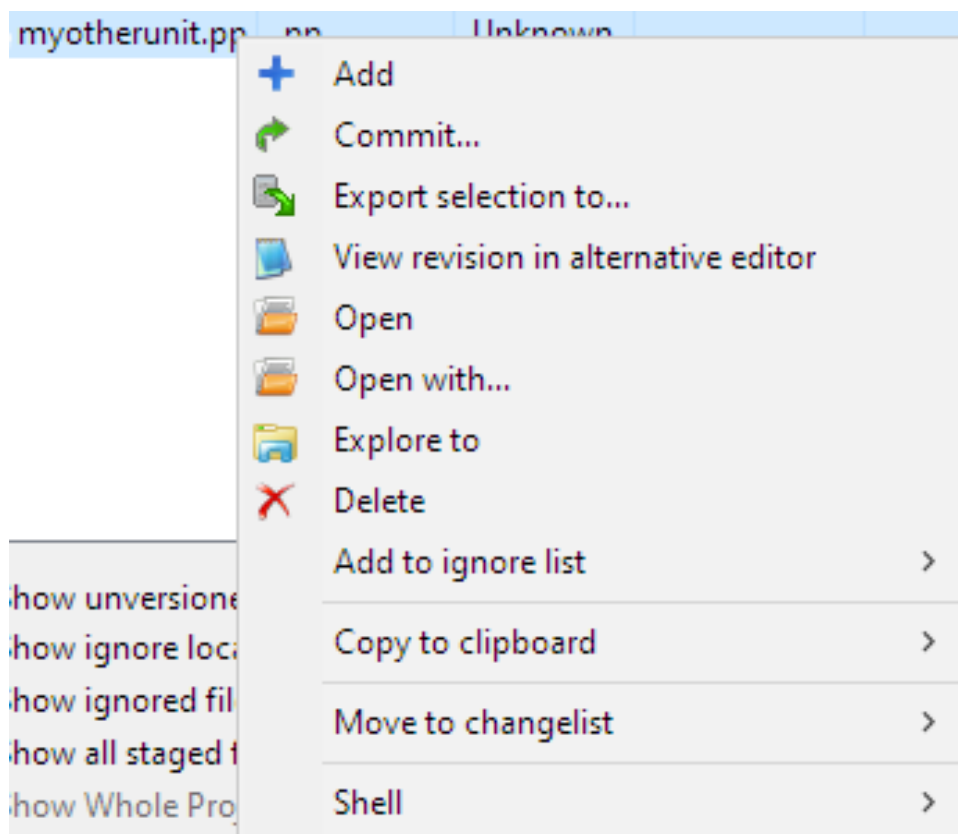
The list of files after `Changes to be committed` is the change that Git will record when you actually commit the change in the next step. This list of changes is often called the `INDEX`.

The process of adding a file can be done for as many files as you want: as long as you do not commit the changes, the change is not complete, i.e. it is not yet recorded.

To do this in TortoiseGit, the context menu of the file explorer can be used:



Or the context menu of the status dialog:

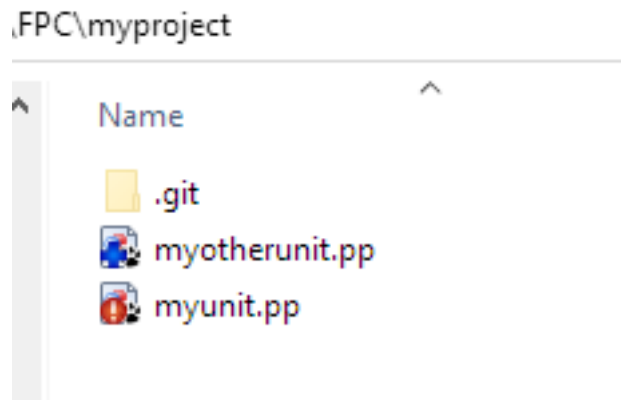
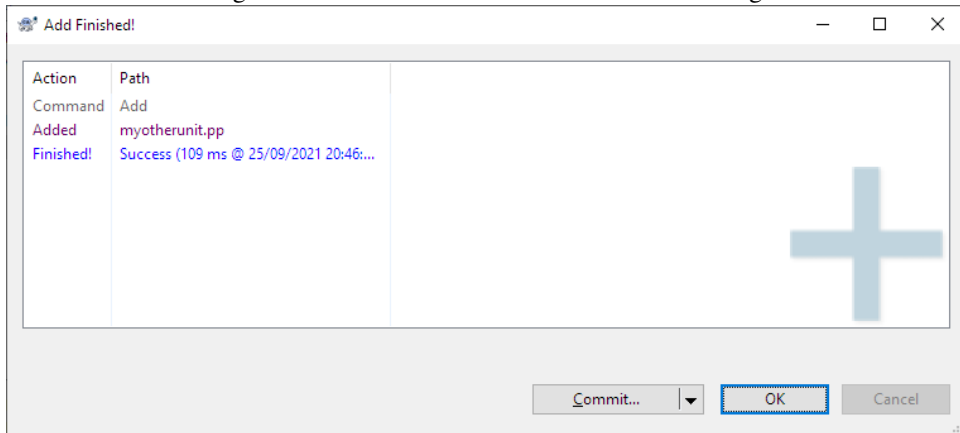


In fact, in any TortoiseGit dialog that shows a list of files, the context menu can be used to do some common Git operations.

Once you chose Add, TortoiseGit will confirm that the file has been added, see figure 2 on page 5.

You can also see it in the explorer window:

Figure 2: TortoiseGit file added confirmation dialog



4 Recording a change: Adding a changed file

The `git status` command showed that a modified file (`myunit.pp`) was present. To record the change in this file, it must also be added to the commit.

```
git add myunit.pp
```

This tells Git that you wish to record the changes of the file in its current state. When you run `git status` again, you'll see the following:

```
> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
new file:   myotherunit.pp
modified:   myunit.pp
```

In TortoiseGit, there is nothing to do. There, all actions are performed when committing.

5 Committing the change

When you're done recording changed items, it is time to finalize the recording of the changes. This is done with the `commit` command:

```
git commit -m 'Some new hash function'
```

The `-m` command-line option allows you to enter a commit message.

If you do not use this option, Git will automatically invoke your editor program and you can write a commit message there. It is good practice to enter a descriptive commit message: it helps developers to search for a particular change.

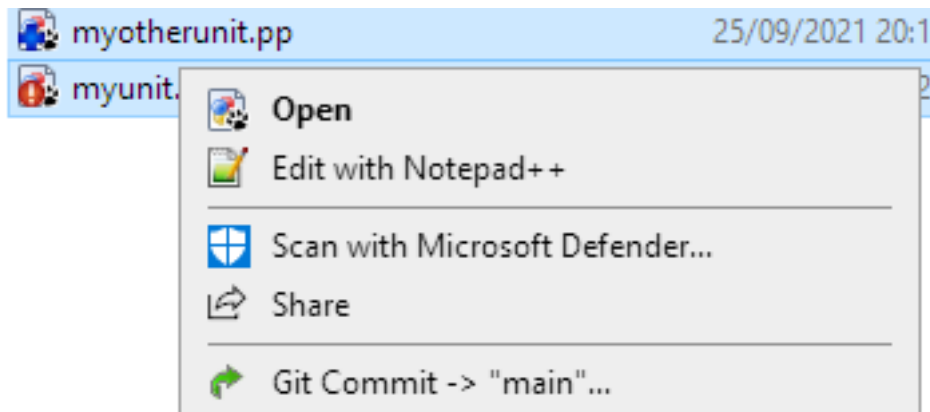
Git will reply with a summary of the actions it did:

```
[master 2398228] * Some new hash function
 2 files changed, 23 insertions(+), 1 deletion(-)
 create mode 100644 myotherunit.pp
```

After this, the status of all files will be reported as up-to-date:

```
> git status
On branch master
nothing to commit, working tree clean
```

In TortoiseGit, committing happens in the commit dialog, which is invoked from the explorer context menu:



It can also be invoked from the 'Check modifications' dialog shown earlier. When selected, the dialog shown in figure 3 on page 7 pops up. In this dialog you can easily select which files need to be committed, and if need be you can still add files that were not yet added to the change to be committed: by checking the 'Show Unversioned Files' checkbox, untracked files can also be shown.

In the dialog, you can add the comment to the commit, and by pressing the 'Commit' button, the commit will be performed. If the commit is successful, TortoiseGit will show a confirmation dialog, which shows basically the same information as you would see on the command-line. An example is shown in figure 4 on page 7

Figure 3: TortoiseGit commit dialog

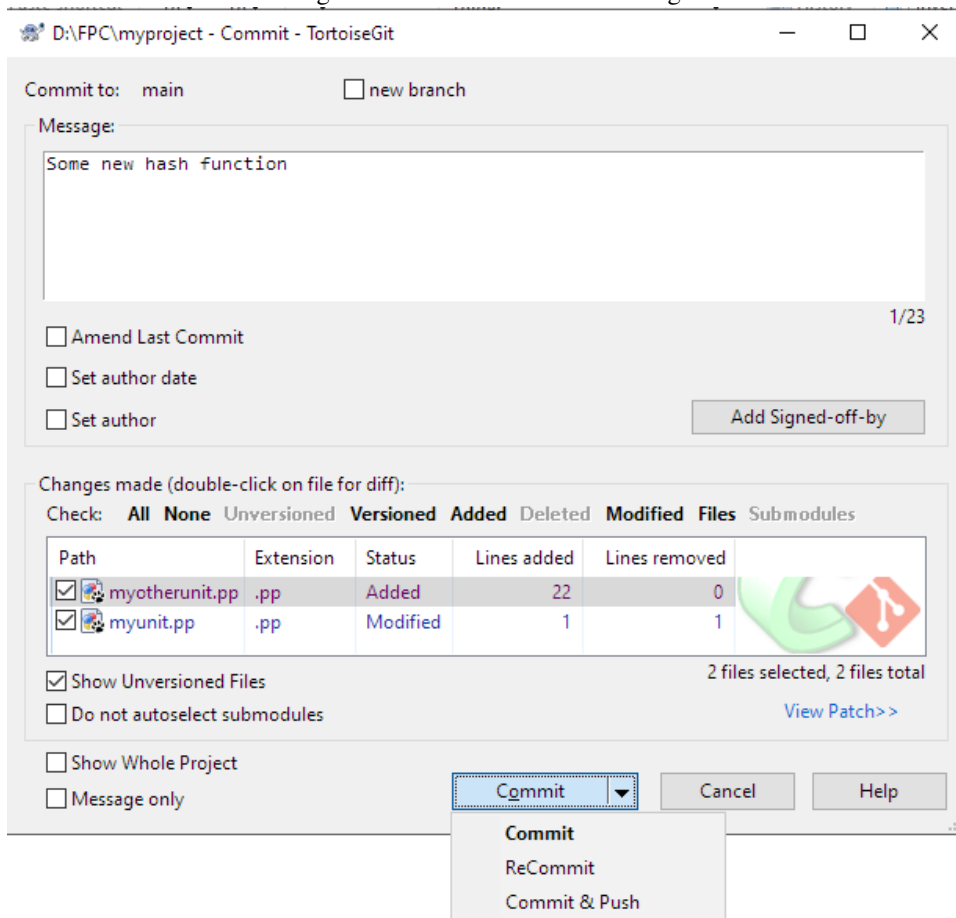
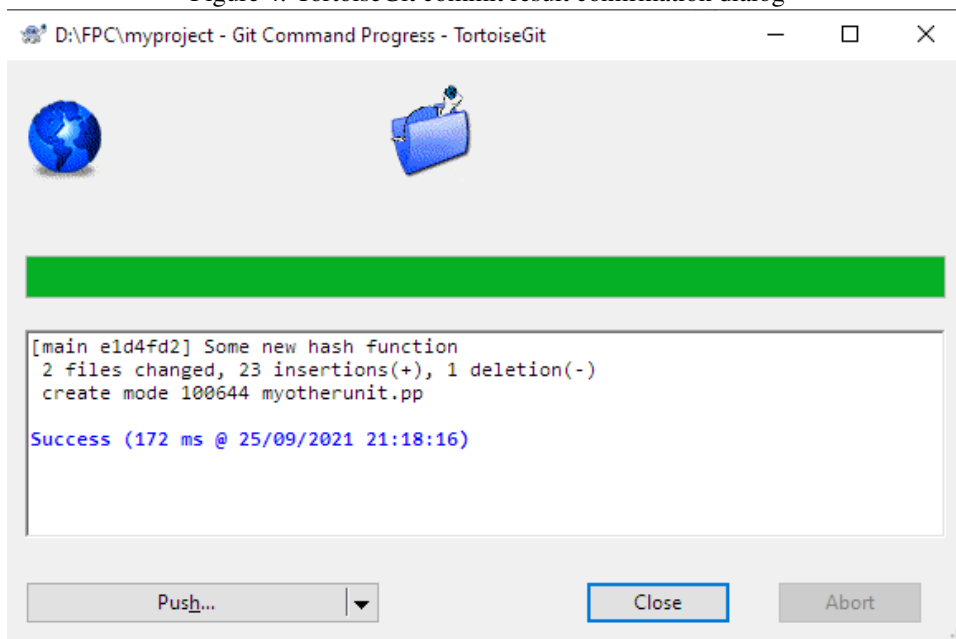


Figure 4: TortoiseGit commit result confirmation dialog



6 Sending changes to a remote repository: Push

After the commit operation, the changes have been recorded locally: your local repository contains a new version of the files. Because Git is a distributed version system, every copy of the repository exists as a repository on its own. That means that when you commit a change, this change is effectively only recorded in your own copy of the repository.

But if you cloned the local repository from a remote repository and wish to contribute your changes to the project, you will still need to send the changes which are recorded and stored in your local repository to the remote repository.

This operation is called ‘pushing’ your changes to the server, and the command for it is `git push`. This will compare the list of changes recorded on the remote repository with the list of changes recorded locally, and sends to the remote repository all local changes that do not yet exist on the remote repository.

How do you know where the changes will be pushed to? The `git remote` command will tell you this:

```
> git remote -v
origin git@gitlab.com:mvancanneyt/myarticleproject.git (fetch)
origin git@gitlab.com:mvancanneyt/myarticleproject.git (push)
```

In this output, `origin` is the name of the remote repository. This name is created automatically when you clone a remote repository.

Because Git is a distributed version control system, there can be multiple remote repositories: each will have its own name. In that case, the `remote` command will list all remote repositories:

```
> git remote -v
ondrej https://gitlab.com/onpok/fpc.git (fetch)
ondrej https://gitlab.com/onpok/fpc.git (push)
origin git@gitlab.com:freepascal.org/fpc/source.git (fetch)
origin git@gitlab.com:freepascal.org/fpc/source.git (push)
```

The pull and push URLs can differ in theory, that is why git shows 2 URLs for each remote.

A second (and probably more likely) question is of course: Exactly what changes will be sent to the remote server when you push? For this you can use the `git log` command:

```
git log origin/main..HEAD
```

The above command lists all commits between the remote revision `origin/main` and the local `HEAD` revision: these are aliases for the last (locally known) commit on the `main` branch on the server and the last local (`HEAD`) commit.

If you have multiple remote servers configured, you simply need to replace the `origin` with the correct name of the remote repository.

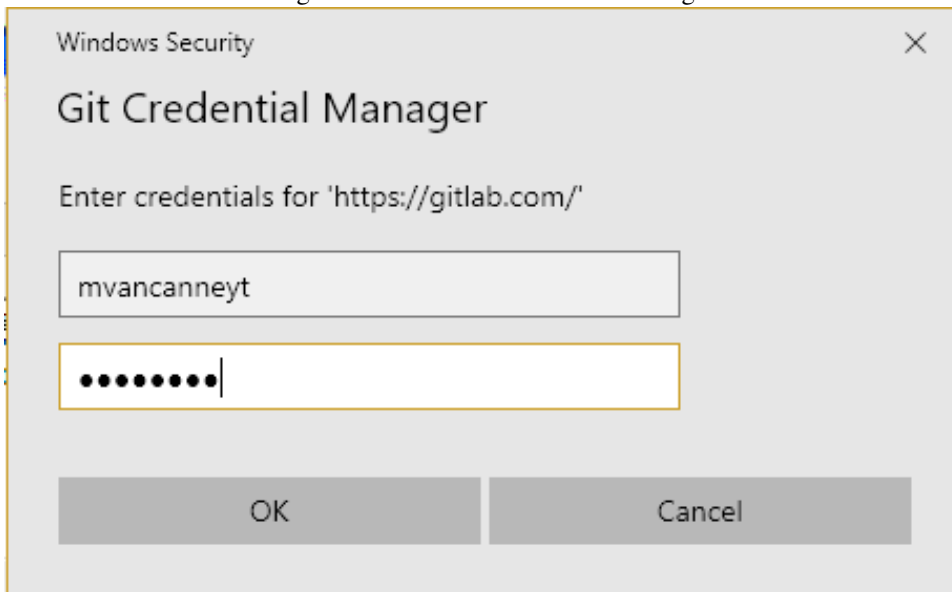
Sending the change is done with the `push` command of Git. It’s quite simple:

```
git push
```

If all goes well, then Git will print some diagnostic messages, telling you what it is doing:

```
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
```


Figure 5: TortoiseGit credentials dialog



```
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 918 bytes | 918.00 KiB/s, done.
Total 7 (delta 1), reused 0 (delta 0)
To gitlab.com:mvancouver/myproject.git
   4fcf99a..8143d7d  main -> main
```

If you wish to push only to a single remote, you can give the name of the repository:

```
git push ondrej
```

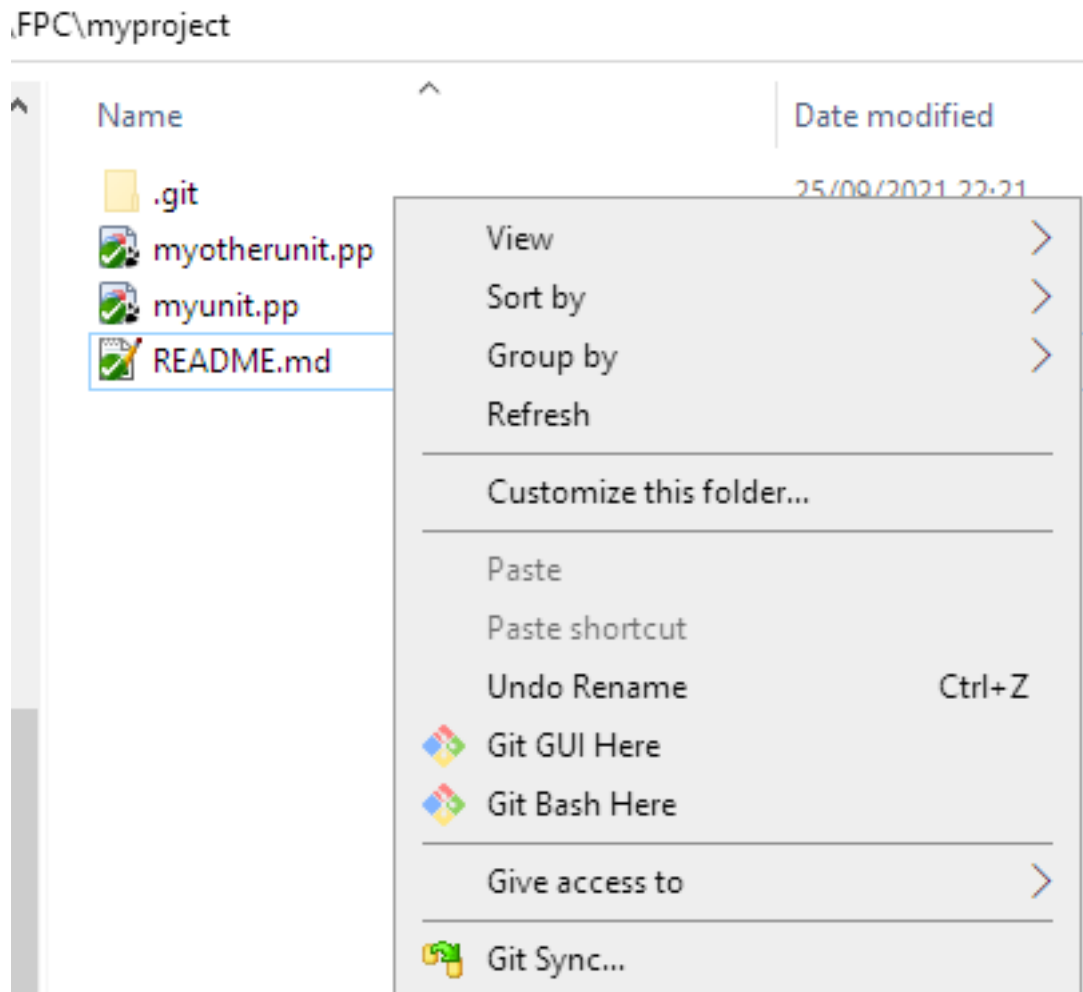
If no repository is specified, then Git will assume 'origin' as the name.

Depending on the configuration and where you cloned from, Git will ask you for a username and password if the remote repository requires you to be authenticated: for Gitlab and Github this will definitely be the case. Git supports several transport mechanisms: HTTP(s) and ssh are among the most used.

ssh In this case, no credentials will be asked: Git will use the ssh keys that you configured for your SSH setup.

https Here, the credentials must be entered once when you connect to the remote repository. (see figure 5 on page 9 for an idea of what this looks like in TortoiseGit) By default Git (or TortoiseGit) will then save them so they will be used the next time you need to authenticate.

To push changes to a remote repository in TortoiseGit, the 'Sync' menu item in the Explorer's context menu must be used.



When selected, the synchronization dialog pops up, it looks like figure 6 on page 11. This dialog allows you to synchronize your local repository with a remote repository: It allows you to select the repository to which you will push changes and it shows the list of local changes that will be pushed to the selected repository: this is essentially the output of the `git log` command mentioned earlier.

There are some other options, but we will not discuss them at this point.

For the push operation, the `Push` button below the list of changed files must be used. If all goes well, then TortoiseGit will show the result of the operation, just as it is shown when pushing on the command-line, see figure 7 on page 11.

It can happen that the push operation fails: if the remote repository has received changes from other contributors which are not yet present in your local repository, then the server will refuse to apply your changes. In that case you will first need to pull the changes from the remote repository, and when that was successful, you can attempt to push your changes again.

That is also the reason why TortoiseGit has a single ‘synchronization’ dialog and not separate dialogs for push and pull: The 2 operations often must be executed one after the other, and this is easier when the operations can be performed from a single dialog.

Figure 6: TortoiseGit synchronization dialog

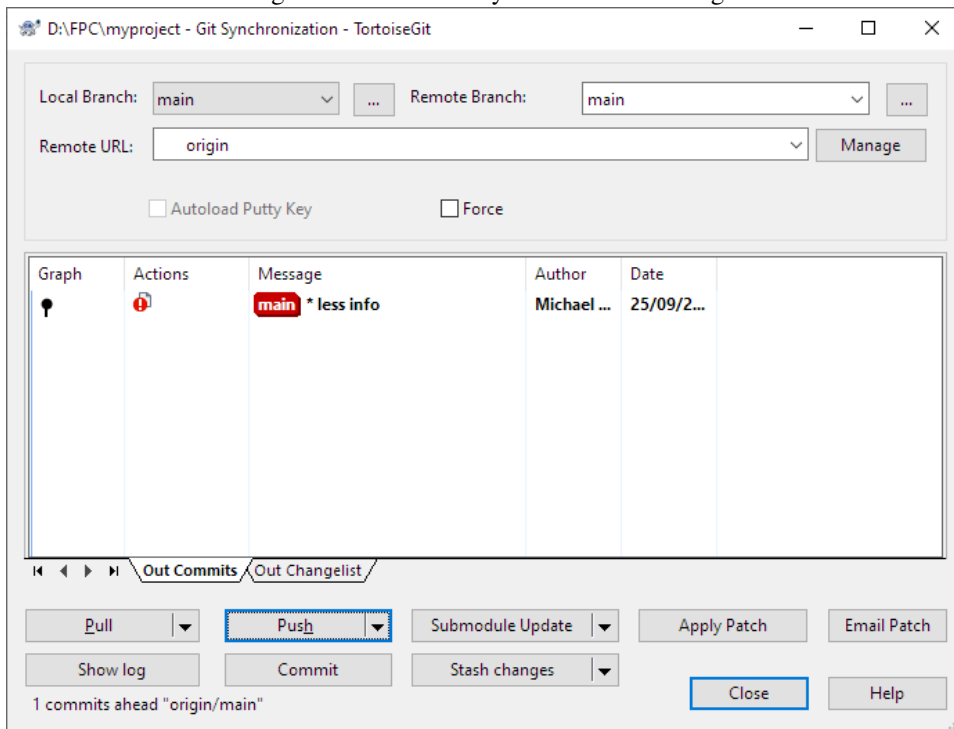


Figure 7: TortoiseGit synchronization result confirmation dialog

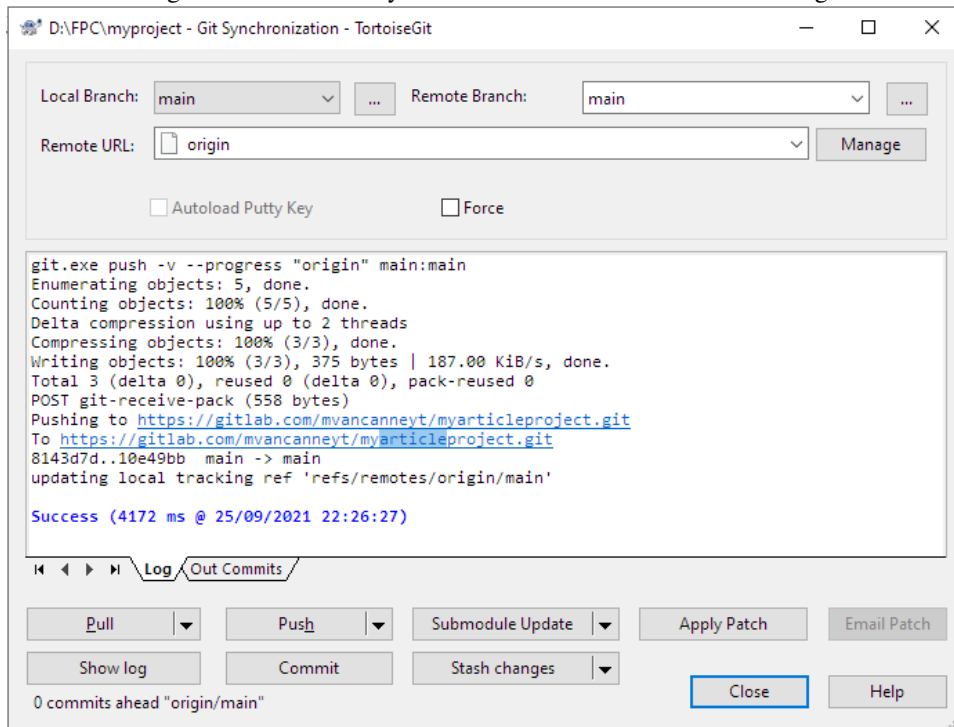
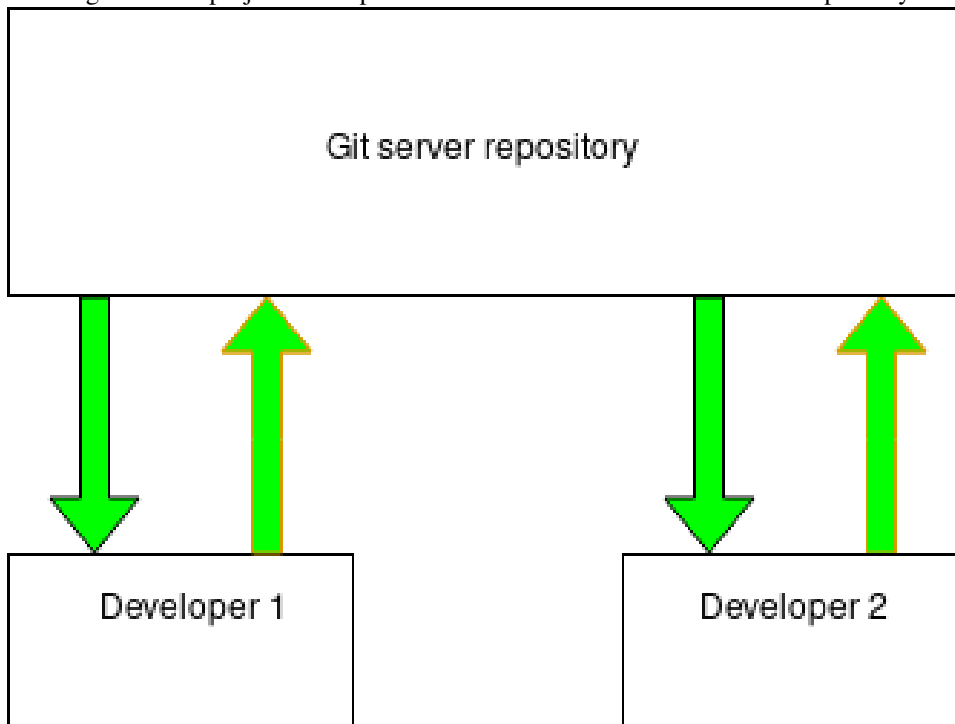


Figure 8: All project developers have read-write access to the remote repository



7 Collaboration: Using Forks

When you clone a repository from Gitlab or Github to your local machine, chances are that you have done so for a repository that you don't have write access to: most projects do not allow arbitrary developers to push changes directly back to their hosted repository. Instead, only a limited amount of people will have write access to the repository on Gitlab or Github, a situation depicted in figure 8 on page 12.

So if you don't have write access to the remote repository, how can you contribute your changes to the project?

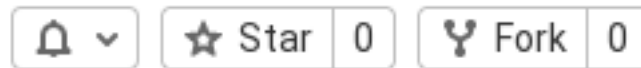
The TortoiseGit 'synchronization' dialog (figure 6 on page 11) shows one possibility: The 'Email patch' button can be used to create a patch file and mail it to one of the developers of the project. The project developer can then use the 'Apply patch' button to apply your patch to his or her local repository and push it to the remote repository: he will presumably have the necessary authorizations to perform the push.

This of course implies that you must know the email address of one of the developers.

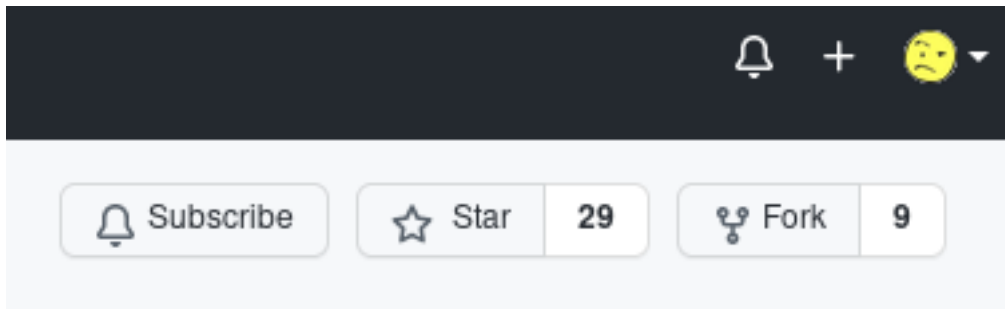
However, the creators of Gitlab and Github have created an easier way to send your changes back to the project repository: *Forks*.

Git is a distributed version system: this means that there can exist many copies of a repository, and they can all be kept in sync in a more or less automated manner. What sites as Gitlab and Github do is to allow you to create a copy of a repository on their servers: this copy is called a *fork*.

In Gitlab, this is done simply by clicking the 'Fork' button in the project page of the project you wish to fork:



A similar button exists in github's project page:



In order to create a fork, you must have an account on Gitlab or Github. The copy will be registered under your account, and the developers will see that you have created a fork. In difference with the original project repository, you will have full control over the fork.

The situation and possibilities are depicted in figure 9 on page 14.

- You can clone your copy locally, and write from your local copy to your forked repository on the server (the 2 green vertical arrows at the right).
- You can also set up your fork to automatically pull changes from the original project repository, so you will always get the latest changes from the original repository (the dark blue arrow at the top).
- You can even register 2 remote repositories in your local copy: your fork and the original repository. If you do this, you can manually keep your forked repository up to date: you pull changes from the original project repository (the light-blue arrow pointing down-right), and push them to your fork (green up arrow).

So how can you contribute changes in this scenario? This is depicted using the orange arrow in figure 9 on page 14. The way to send back your changes is called a merge request.

When you push changes to your fork, the Gitlab or Github servers can be configured to automatically send you an URL which you can use to create a merge request. This URL will take you to their website, where you can fill in the details of the merge request. For Gitlab, the merge request page looks like figure 10 on page 14.

You select the source repository (your fork) and branch (main, or the branch where you committed your change) and target repository (this is normally the project repository) and

Figure 9: Flow when using a fork

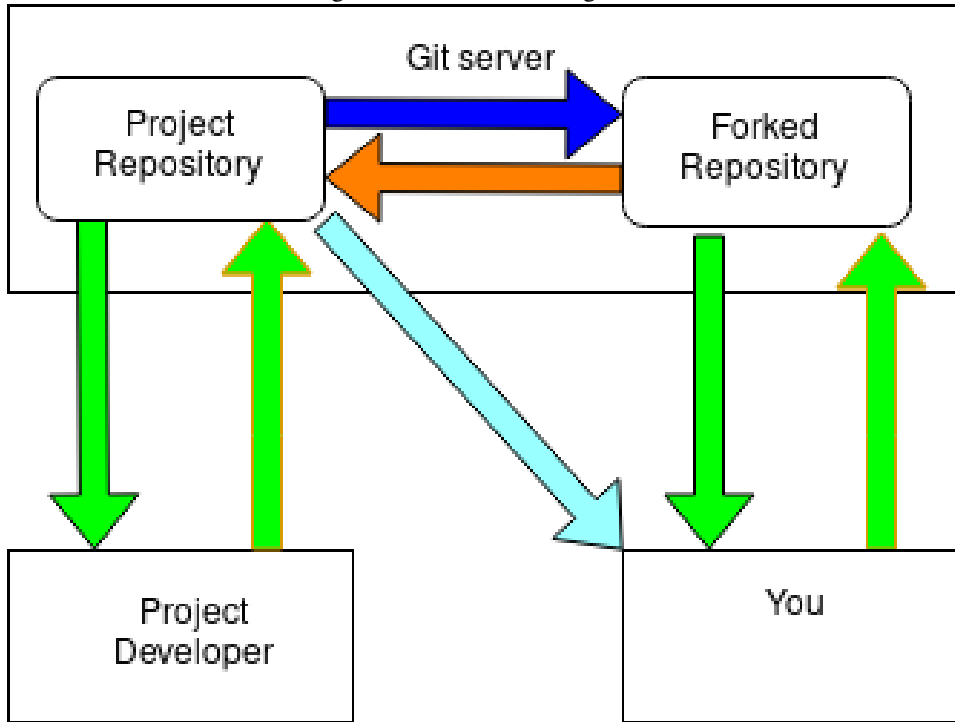


Figure 10: Starting a merge request

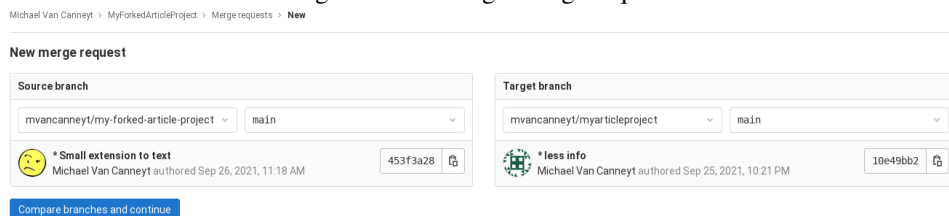


Figure 11: Provide info about your request

New merge request

From `mvancanneyt/my-forked-article-project:main` into `mvancanneyt/myarticleproject:m`

Title

Start the title with [Draft:](#) to prevent a merge request that is a work in progress. Add [description templates](#) to help your contributors communicate effectively.

Description

Write Preview

Describe the goal of the changes and what reviewers should look for.

[Markdown](#) and [quick actions](#) are supported

branch where the change is supposed to go. When you continue, a dialog appears (figure 11 on page 15) where you can give the developers of the project an explanation of what your patch does, what problem it solves, or why you think it is a necessary or useful change: in general, provide any info to help the developers decide whether or not they want to incorporate your change into the project.

It is worth noting that although you will be creating the merge request starting in your forked copy of the project repository, the actual merge request will end up in the original project. You will not see it in the list of merge requests of your project.

For Gitlab, all this can be done on the command-line as well, but the process is rather involved and requires extensive command-line skills.

A merge request is just what the name says: a *request* to merge a change. This has several consequences:

- The change is not automatically merged into the repository of the developers.
- Instead, it is recorded in the Gitlab or Github platform.
- The developers are notified of your request: they receive an email with the details of your request.
- The request is shown to them when they go on the website (see figure 12 on page 16 for how this looks for a developer of a project on gitlab).
- They can examine your changes using the website, or check out your version of the files.
- The developers can create comments on your merge request, add comments to details of the diff (figure 13 on page 17 shows what a developer sees from your merge request).

Figure 12: The project developers see a list of merge requests



* Small extension to text

!1 · created Sep 26, 2021, 11:23 AM by Michael Van Canneyt

- The developers can request additional changes: if you make additional changes in the same branch and push them to your forked repository they will automatically be added to the merge request.
- There can be an approval process.
- Automated tests can be run.

There is a large list of possibilities: a whole ecosystem of options exist to help the project developers to examine and process your request.

If all went well, the project developers decide to accept your merge request. The merge request is then approved (although this is optional) and the change is actually merged into their repository: this can be done in 2 ways:

- Using the button in the website. In that case, the request looks like figure 14 on page 18.
- By applying the changes on a local repository and pushing them to the server. Depending on the way this is done, the merge request will also be closed automatically.


The setup of platforms like Gitlab and Github offer a lot of configuration to automatically handle merge requests. If CI (Continuous Integration) was set up, then they can for example configure the server to automatically run a test suite on the sources of every merge request and even allow your merge request to be merged automatically if the test suite runs without fail. The details of this depend of course on the service level of your project account and the hosting platform you are using.

8 Conclusion

In this article, we have shown how to communicate changes to a remote repository, and how this can even be done when you do not have write access to the project's repository: Forks are the feature of platforms such as Gitlab and Github which make Git such a successful collaboration tool. But we have not yet covered all of Git - not by a long shot. In the next Git article, we'll cover the use of splitting large changes into smaller commits and the use of branches in Git, as it is also a major topic when co-operating on a project managed by Git.


Figure 13: Details about the merge request


Michael Van Canneyt > MyArticleProject > Merge requests > !1

Open Created Sep 26, 2021, 11:23 AM by  **Michael Van Canneyt** Maintainer

* Small extension to text



Overview 0 Commits 1 Changes 1


 Request to merge `mvancanneyt:main` into `main`

 **No pipeline** [Add the .gitlab-ci.yml file](#) to create one.

Are you adding technical debt or code vulnerabilities?
Use [CI pipelines to test your code](#) by simply adding a GitLab CI configuration file to your project. It only takes a minute to make your code more secure and robust.

Show me how to add a pipeline

 Approve Approval is optional 

 Merge

> **1 commit** and **1 merge commit** will be added to `main`. [Modify merge commit](#)





 0  0 


Figure 14: The merge request was accepted and merged


Michael Van Canneyt > MyArticleProject > Merge requests > !1



Merged Created Sep 26, 2021, 11:23 AM by  **Michael Van Canneyt** Maintainer

* Small extension to text



Overview 0 Commits 1 Changes 1

 Request to merge `mvancouver:main` into `main`

 Approval is optional

 **Merged by**  **Michael Van Canneyt** Sep 26, 2021, 11:25 AM Revert Cherry-pick

The changes were merged into `main` with `318401f3`

 0  0 