CHAPTER X

SOURCE CODE MANAGEMENT, GIT AND GOGS

Table of content

Introduction 7
The Server to Client model 7
Branching 10
The merge processes. 14
Different scenarios. 18
Repositories. 19
Our new feature and misconception scenario. 20
A SCM Tool, summary 23
Git in a nutshell 26
GOGS 28
Gogs installation 28
Introduction 28
Entities & relationships. 34
Continuing the tour. 34
..... 34
Creating a Gogs repository. 39
Let's create our first Gogs repository. 42
Starting the game 45
Introduction 45
Creating a first repository 54
Connecting to an existing repository 55
The fork process. 56
MS Visual Studio Client 58
Appendices 59
Gogs Installation. 59
Gogs log, and error log files 60
Connect the uninitialized Gogs repository to a new local one. 61
Connect the Gogs repository to an existing local one. 81
The fork process. 90
The MS Visual Studio client. 107
The git communication protocol 118

Figure 1: The Server Client SCM based system. 8
Figure 2: Consistency of the source code package. 9
Figure 3: Branching the tree. 10
Figure 4: an SCM client version 1. 12
Figure 5: The SCM client is an IDE extension. 13
Figure 6: Integrating master branch into the feature branch (reconciliation). 15
Figure 7: Involving the feature branch team in the misconception. 17
Figure 8: An SCM repository symbol. 19
Figure 9: Feature and misconception branch scenario A. 21
Figure 10: The final master branch. 24
Figure 11: Git collaborating repositories. 26
Figure 12: Launch the Gogs web page. 28
Figure 13: Gogs sign in. 29
Figure 14: Gogs main functions. 29
Figure 15: the Dashboard. 30
Figure 16: Gogs issues 30
Figure 17: Discussion on an issue. 31
Figure 18 Pull Requests. 31
Figure 19: Exploring Gogs. 32
Figure 20: Users, Teams, Organizations & Repositories. 33
Figure 21: Accessing the admin panel. 34
Figure 22: The admin panel. 35
Figure 23: User Manage Panel 35
Figure 24: Main dashboard panel. 36
Figure 25: Organization sub-panel. 37
Figure 26: Organization Panel. 37
Figure 27: Invite someone to an organization. 38
Figure 28: Organization Owners. 38
Figure 29: Different collaborating repositories. 40
Figure 30: Create a first repository. 42
Figure 31: MyFirstRepository. 43
Figure 32: First uninitialized repository. 44
Figure 33: Git Bash on NewFeature branch. 47
Figure 34: Git push --all pushes all branches. 47
Figure 35: Difference between git fetch and git pull. 48
Figure 36: Content of MyFile01.txt in NewFeature branch. 49
Figure 37: Content of MyFile01.txt in the master branch. 50
Figure 38: A merge of NewFeature in the master branch. 50
Figure 39: Editing MyFile01.txt in merge state. 51
Figure 40: The two versions reconciliated. 51
Figure 41: NewFeature reconciliated with master. 52
Figure 42: Back to final state. 52
Figure 43: Git commands moves summary. 53
Figure 44: Pull Request from a collaboration. 56

Figure 45: MS Visual Studio Command Prompt. 58
Figure 46: Open A git Client. 61
Figure 47: Git Bash Command prompt. 62
Figure 48: Git init command. 62
Figure 49: README file creation. 63
Figure 50: Create a first project file. 63
Figure 51: Adding files to the project. 64
Figure 52: Feeding the repository with (a new version of) files. 64
Figure 53: Git file transfer. 65
Figure 54: Gogs Repository main screen. 66
Figure 55: Tags related information. 67
Figure 56: Changing a file. 68
Figure 57: commit and push the change. 68
Figure 58: pushed change. 68
Figure 59: file changed in Gogs. 69
Figure 60: File History. 69
Figure 61: Launching Git GUI 69
Figure 62: Git GUI. 70
Figure 63: Third file version 70
Figure 64: Changing a file via Git GUI. 71
Figure 65: Git GUI - push the change. 72
Figure 66: Push Popup. 72
Figure 67: Create a branch from Git GUI. 73
Figure 68: Change a file in the new branch. 73
Figure 69: Push the branch. 74
Figure 70: You see now two branches. 75
Figure 71: The NewFeature branch in Gogs. 75
Figure 72: Changed file in the new branch. 76
Figure 73: The compare button. 76
Figure 74: Compare branches, create Pull Request. 76
Figure 75: Pull request creation. 77
Figure 76: Pull request created. 77
Figure 77: Pull request pending. 78
Figure 78: back to the pull request. 78
Figure 79: branched file in the master. 79
Figure 80: check out back to the master. 79
Figure 81: update the local master branch. 80
Figure 82: local merge. 80
Figure 83: Git Fetch all. 81
Figure 84: Launch Git Gui in a new folder. 82
Figure 85: Change a file in a new copy. 83
Figure 86: Staging a change. 83
Figure 87: Commit the staged change. 84
Figure 88: Push the change. 84
Figure 89: Change in the origin. 85
Figure 90: No change in the NewFeatureBranch. 85
Figure 91: Compare. 86

Figure 92: Switch to a branch. 86
Figure 93: Merge the master into the current branch. 87
Figure 94: NewFeature has been updated. 87
Figure 95: Update a file in the current branch. 87
Figure 96: Push the branch to the remote. 88
Figure 97: File updated in Gogs. 88
Figure 98: Delete the remote branch. 89
Figure 99: One remaining branch after push and delete. 89
Figure 100: Collaborators organization. 90
Figure 101: Connect to the collaborators organization. 90
Figure 102: Team associated to the organization. 91
Figure 103: Collaborators team. 91
Figure 104: Add members to the collaborators team. 92
Figure 105: The collaborators team. 92
Figure 106: The main repository. 93
Figure 107: Repository population. 94
Figure 108: Drag & drop to the main repository. 95
Figure 109: Commit the file addition. 96
Figure 110: The repository populated. 97
Figure 111: Fork the main repository. 98
Figure 112: Main and collaborative repositories. 99
Figure 113: The collaborative repository. 99
Figure 114: cloning the collaborative repository. 100
Figure 115: Create a NewFeature branch. 100
Figure 116: Modify MyFile01.txt on the NewFeature branch. 101
Figure 117: Push the NewFeature branch to the origin. 102
Figure 118: Push popup. 102
Figure 119: Push popup, suite. 103
Figure 120: The NewFeature branch pushed to the collaborative repository. 103
Figure 121: Comparison between the two branches. 104
Figure 122: Pull request creation. 105
Figure 123: Benoit is assigned to the pull request review. 105
Figure 124: Merge pull request. 106
Figure 125: Merged change. 106
Figure 126: ASimpleApp repository. 107
Figure 127: Create a C# desktop application. 108
Figure 128: ASimpleApp C# application. 108
Figure 129: ASimpleApp Git repository. 108
Figure 130: ASimpleApp local Git Repository. 109
Figure 131: Git MS VS Settings. 110
Figure 132: Project Git Settings. 110
Figure 133: The pushed Gogs repository. 111
Figure 134: Say Hello button. 112
Figure 135: MS VS indicators. 112
Figure 136: Changes committing. 113
Figure 137: MS VS indicators. 113
Figure 138: Push the differences. 113

Figure 139: Gogs content. 114
Figure 140: Create a branch from MS VS........ 114
Figure 141: MS VS NewFeature branch. 115
Figure 142: Change the message........ 115
Figure 143: Commit the changes........ 116
Figure 144: MS VS indicators. 116
Figure 145: Changes in the NewFeature branch. 117

Introduction

For those readers who already know what Source Code Management (SCM) and Git is, you can skip this chapter, although it might be convenient to read it through very quickly, if you know only Git or services like GitHub, Azure DevOps or JIRA for example.

This first chapter is nevertheless highly recommended to be read, since it introduces in its own terms and concepts the other ones, particularly the one concerning Gogs.

When you want to track the history of a source code package, knowing who changed what piece of code, for what reason and when, which is necessary when you work as a programmer belonging to a team, you need some tool to manage this, in other words a Source Code Management tool.

Historically, the first packages available for doing such a task were (and sometimes still are) tools like Microsoft Visual Source Safe, Subversion (SVN), Concurrent Versions System (CVS) and others.

The Server to Client model

Most of them are based on *Server to Client base paradigm*, that is a server computer actually holds the source code of your package(s), and the programmer who wants to change a piece of code in a particular source code file needs to “check out” this file from the server, modify it on its own computer, test it and return it back to the server (check in):

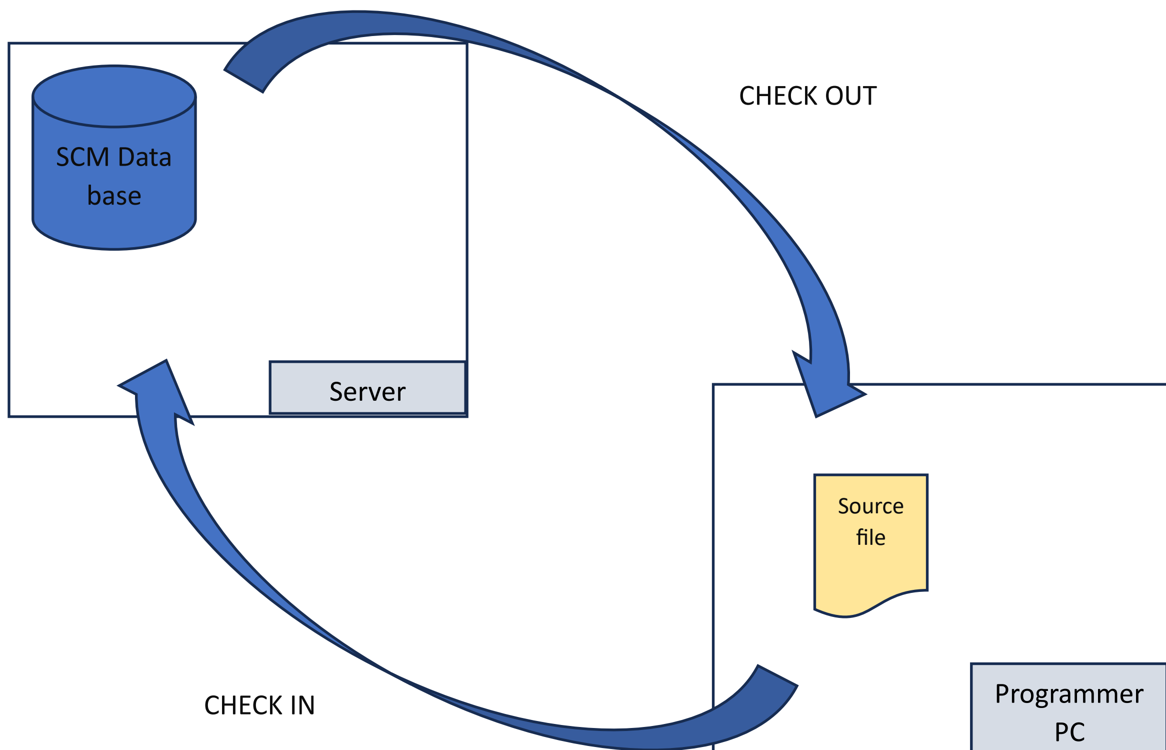


Figure 1: The Server Client SCM based system.

During the check in/ check out process, the file is “locked” by the SCM tool, and no other programmer can check in or out the file(s).

Once checked in (released), other programmers can, at their turn, check out the file and modify it, test it, etc.

Of course, multiple files can be checked out at a time.

During this check out/ check in process, the programmer usually gives information to the SCM system, such as, for example, the reasons for changes (e.g. a bug fix or the addition of a new feature).

Checking out a file prevents other users (programmers) to change the file.

Checking in the file introduces it back to the SCM data base and allows other users to access it again.

The SCM then stores all this information in its database, as well as the part of source code that has been modified (we’ll call *delta* the part of the code that has been changed later in this chapter).

Those check outs/ check in constitute a good way for the SCM to build an history of the whole file package, which the SCM usually shows as “tags” that a programmer can consult, as well as the source code file before and after and before the checkout/ check in.

The programmer is then able to rebuild the image of the package at any point in time.

When a point in time represents a certain release (version) of the package, the SCM often allows to “label” the version at this point in time, allowing then to retrieve the package at a time (version) by

just using the label (the SCM tool is able to provide the complete image of the package at a certain label).

But what happens when different programmers modify different files that are part of the same “functionality”, and this for different reasons (one for fixing a bug, the other for introducing a new feature, as an example)?

Is the source code package still consistent after multiple check outs/ check in from many different programmers?

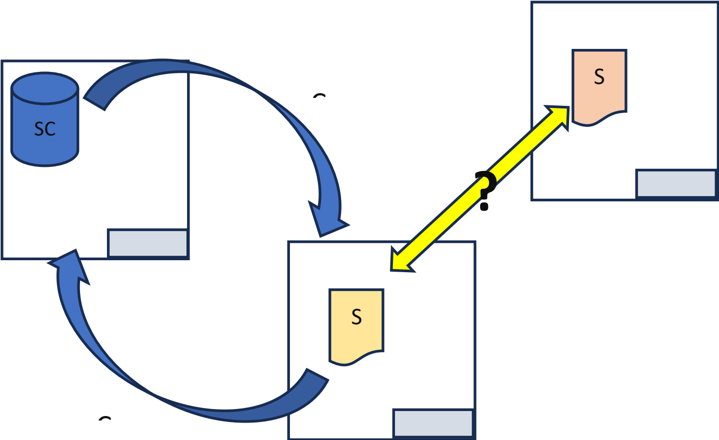


Figure 2: Consistency of the source code package.

Branching

When the purpose of making a team of programmers is to work on a new feature of the package, it would be nice then to create a "branch" of the current package, which is a copy of the current one:

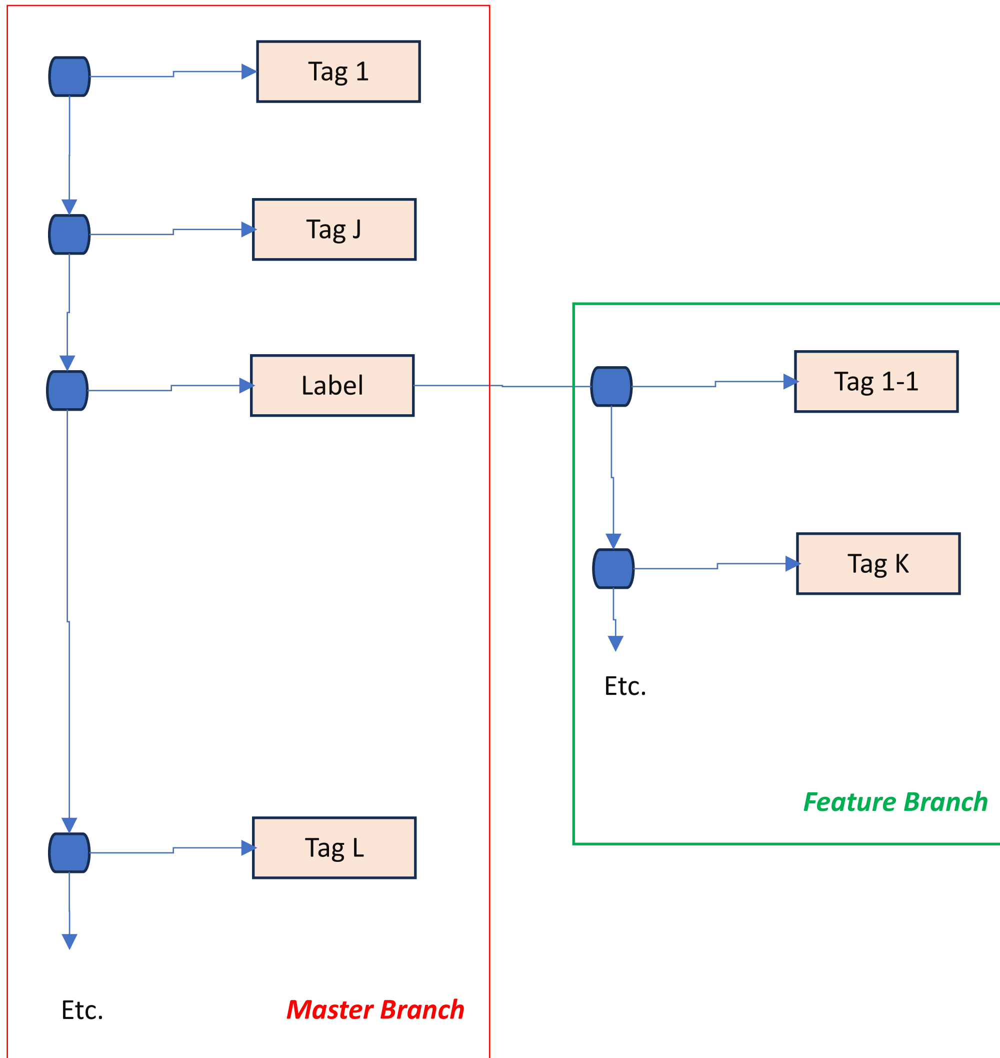


Figure 3: Branching the tree.

The source package the whole programmer team is working on is the *master*.

While the team (or a sub-team) is working on the new feature branch (thanks, for example, to the ability to label a branch, for getting regularly the fresh new version of the feature branch, for the sake of keeping the whole source package code consistent), the (rest of) the team can go on working with the package by fixing bugs for example.[1]

[1] On the other hand-side, a branch can be dedicated to fixing bugs, when it seems urgent to focus more for the team (and its boss!) on the number of unfixed bugs rather than developing new features.

Ok, ok.

But there will be a moment when it will be time to deliver this new feature.

We will have to *merge* the feature branch with the master to deliver a new version.

Several situations may then occur:

- Bugs fixes have no effects on the package functionalities (inconsistencies), no side effect,
- Feature changes have no effects on the master package functionalities,
- Both (such a wonderful world!),
- Bugs fixes and feature changes have touched a set a same source files, but not really affecting functionalities (different part of the code have been changed, for example),
- There are conflicts (same part of code have been changed or changes made on both branches have side effects on the package functionalities).

You will have to “reconciliate” changes made for different reasons.

In any case (and thinking over, regularly in time at each checkout/check-in) there will be a need for a *review* before the merge can occur (done together with the team or a sub team).[2]

On the SCM server, the whole package code consists of the source code and many other information such as the package history, labels, etc.

The SCM server is helpful in describing any change made on the package, but locally, on the developer computer the source code package consists only of the raw source code files, together with the needed development environment (IDE such as Visual Studio, CodeLite, NetBeans, Eclipse and many, many others).

Why not thinking about having a kind of client side SCM on the programmer computer itself?

[2] As we will see further on in this chapter this will be done thanks to a *pull request*.

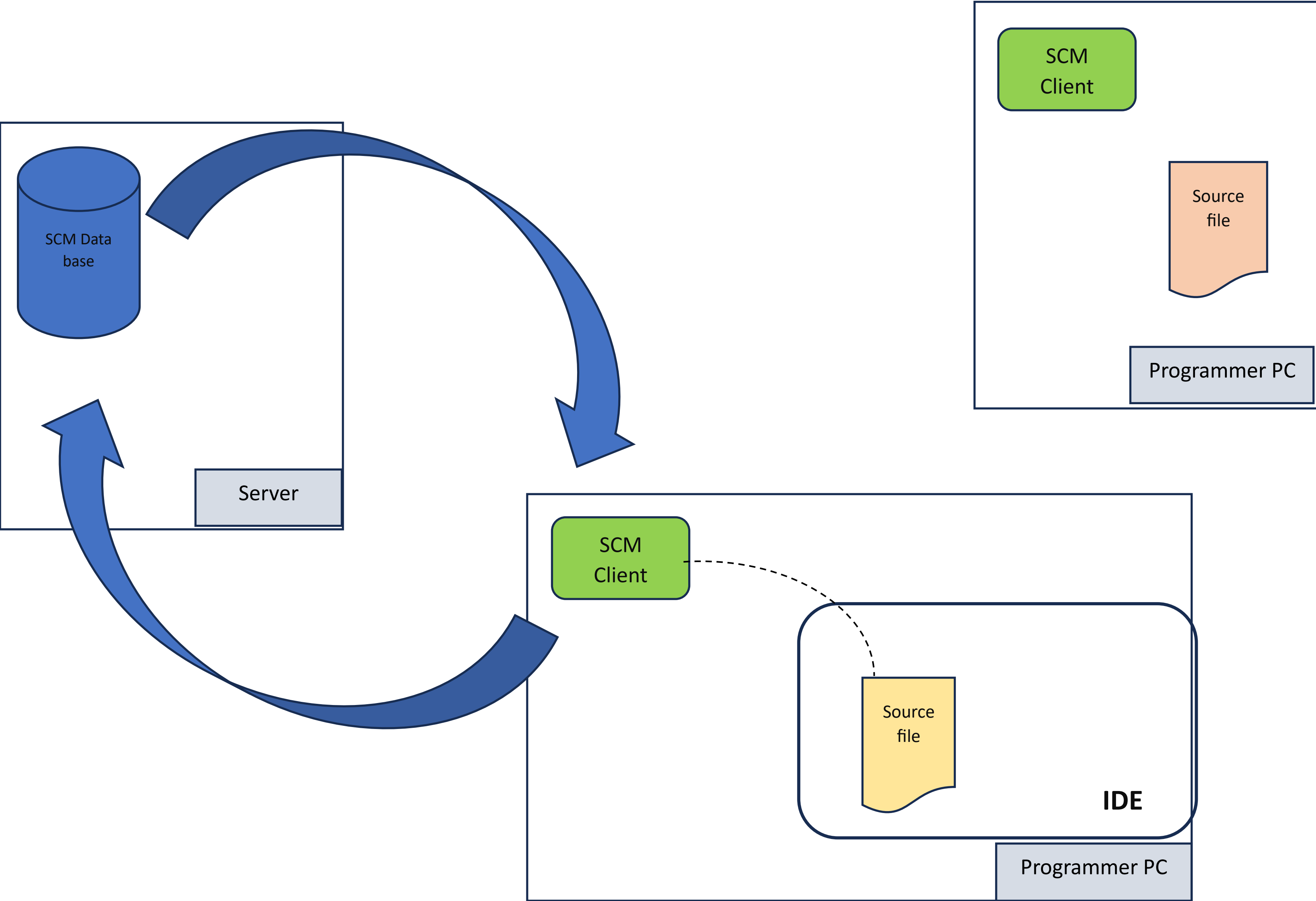


Figure 4: an SCM client version 1.

Or even better (The SCM client side is part - an add-on- of the IDE itself):

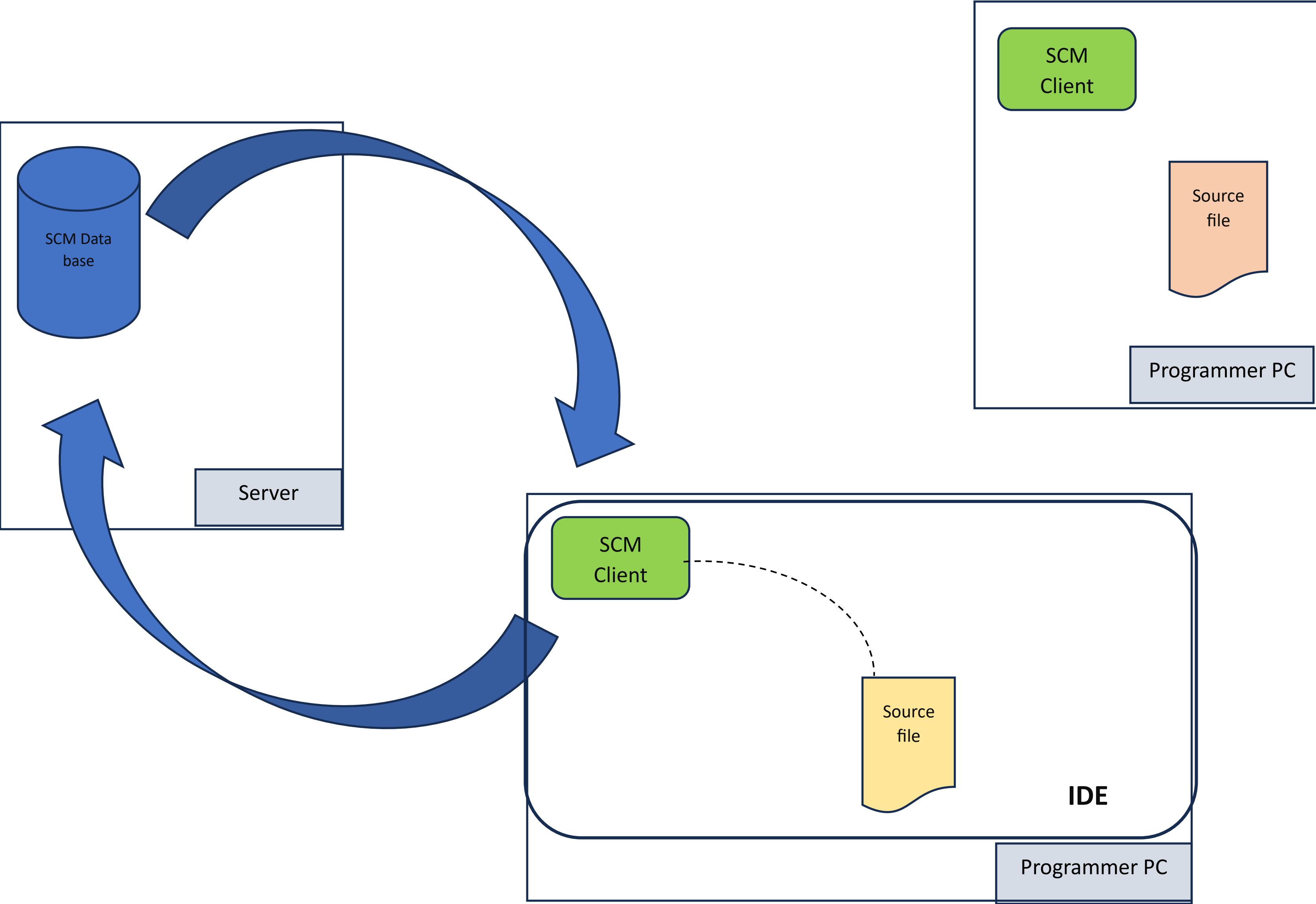


Figure 5: The SCM client is an IDE extension.

The merge processes.

Once the time has come to merge the two branches[3], you'll will have, basically, to review the differences between each and single source file that are different from one branch (master) and the feature branch. But not only. You'll have also to examine possible side effects on both sides[4].

The merge process is not only and necessary a single step process.

Do not forget that the main branch and the other branches may evolve in parallel.

Thus, a good practice, when one or multiple programmers are working on a new feature, is, for the programmers, to regularly integrate the current package source (i.e. the *main* or *master* branch) in the version he is working on locally on the new feature branch on its computer[5].

[3] And, again, at each check in, eventually.
[4] You'll have to examine it from a reviewal and conceptual point of view, but also through the continuous software development life cycle, and in example through regression testing.
[5] Even if multiple files are checked out by other programmers, it does not prevent any programmer to take a copy of the checked file directly from the current "owner" of the file). Alternatively, the file owner can regularly check in and out again files he is modifying, in order to make them available to the rest of the team(s).

This can be a multiple step phase, and you can decide to only integrate a part of the new feature, combined with a part of bug fixes, since during the build of the feature branch it is suggested to regularly integrate some bug fixes into the feature branch, as illustrated below:

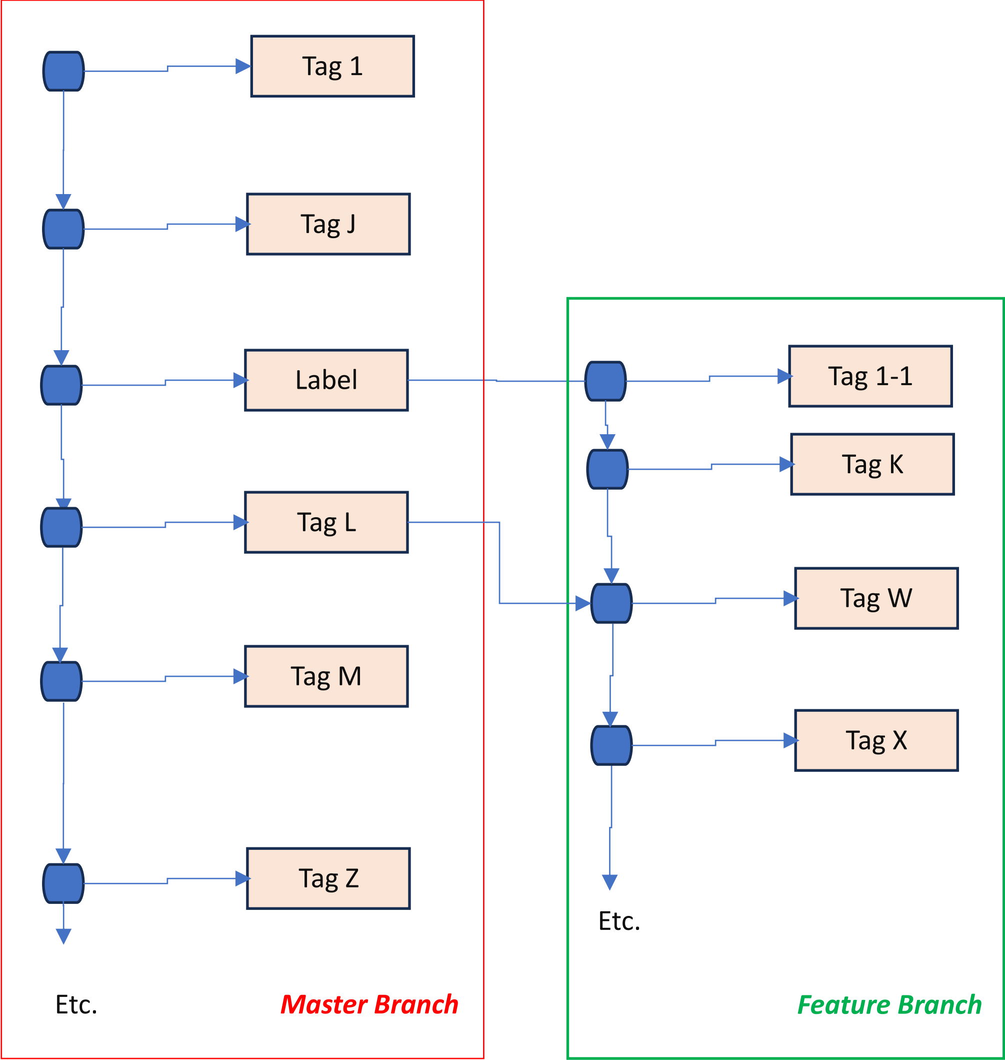


Figure 6: Integrating master branch into the feature branch (reconciliation).

Now, suppose that that bug fix (Tag L in master branch, Tag W in feature branch) put in evidence that there is a misconception (design mistake) in the package, and then, consequently, the same thing in the feature branch.

Much harder, suppose that you, and your team discover that you need additional team expertise discussion about how to solve this misconception.

You could create another branch from Tag L on the master branch. Let's call it "Misconception branch". That is one solution.

You could also think to yourself: "well, since we have a (sub)team dedicated to a new feature, why not asking them also to think about this misconception – involving eventually another team[6].

But, since it is important to go on with the new feature development, you go on also[7] with the feature branch.

And now you can also imagine another scenario, showed on the next page.

[6] That is an example where you'll see the quite important impact of Git, that is the collaboration with other teams, inside, or *outside* of your team or company.
[7] And together.

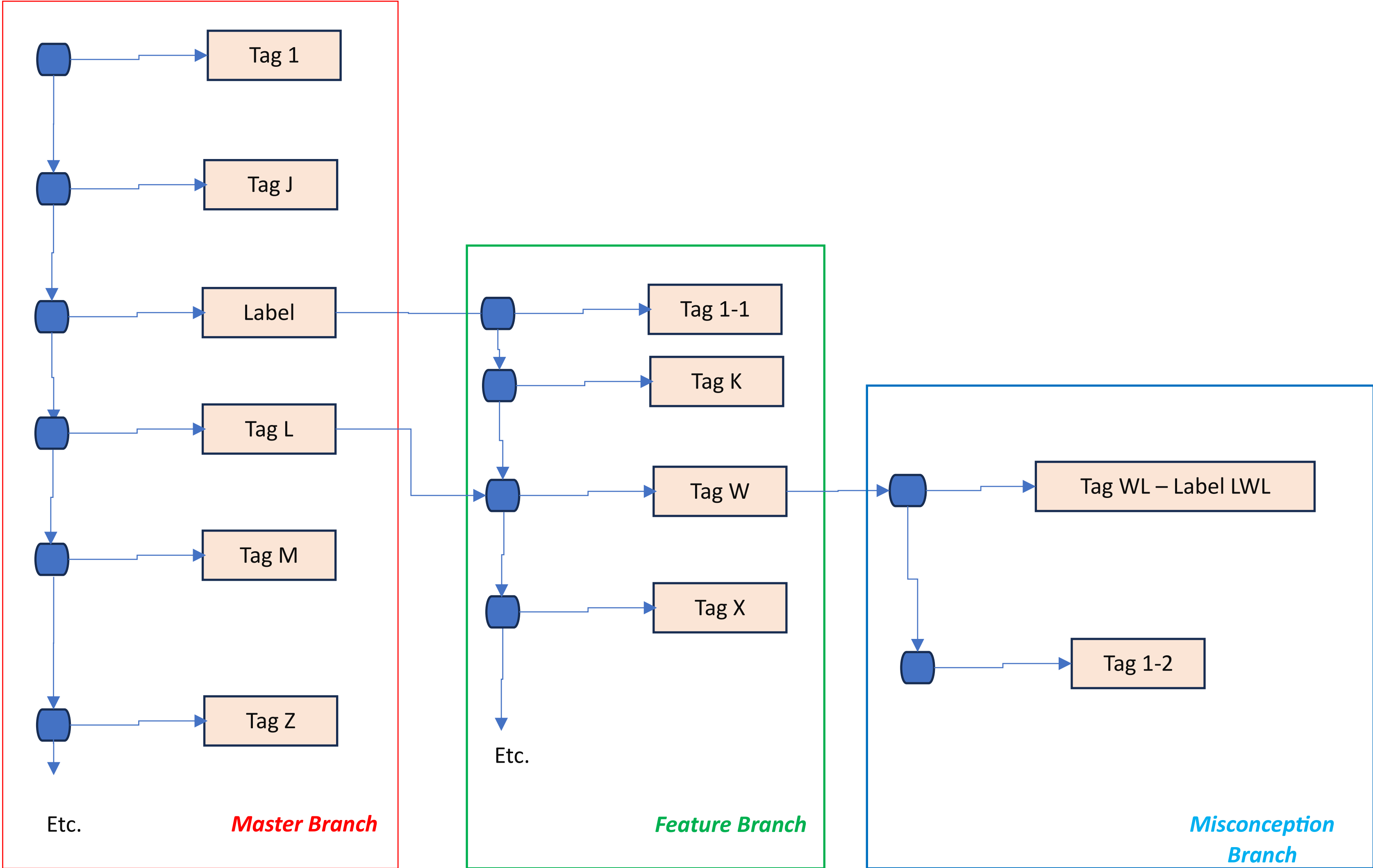


Figure 7: Involving the feature branch team in the misconception.

Different scenarios.

Managing source code is not just a question of using an SCM a or b, if you foresee to work with or together with different teams, sub teams, locally or remotely, belonging to the same or different organizations.

Additionally, you might want to rely only on cloud based or self-hosted systems such as GitHub, Azure DevOps, or JIRA.

On the other hand side, it may be interesting to have locally in your own organization and even on your single computer an SCM or SCM server installed.

In summary the complete source code package development might be based on the use of multiple SCM tools. And multiple teams. And organizations. Concretely, it means:

- You might need multiple collaborative teams and, consequently:
- Multiple collaborative SCM tools

Those teams might use different SCM tools, thus using different merge reviews processes.

The SCM databases might be local or remote, which involves considering the communication protocol(s) the SCM tools are using.

The way you manage change requests (such as introducing a new feature for example), the bugs fixings, might be different.

You might also want to have your own SCM tool, partly or completely developed by your team (after all we are programmers, aren't we?).

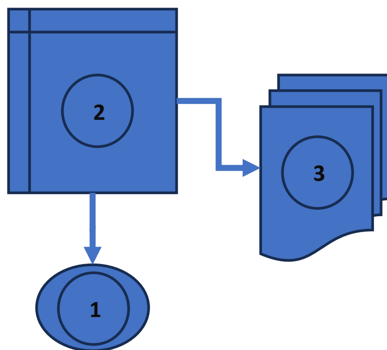
In the context of this book, we'll use Gogs.[8]

---

[8] Gogs is written in GoLang. If you are familiar or even if you are regularly using this language, it is extremely interesting to use Gogs, since you can then adapt and enhance your SCM tool to your needs.

Repositories.

From now on, instead of using the terms SCM *data base*, we will use the term SCM *repository* instead.



This symbol will be used for representing a repository.

1. The SCM repository monitoring engine,
2. The SCM repository hosting structure file(s),
3. The different (source code) raw files, including or not specific files used by the development IDE.

Figure 8: An SCM repository symbol.

The SCM repository monitoring engine is the one responsible for:

- Receiving the checked in files and "pouring" them into the repository hosting structure, together with the tag or label and associated comments, author, and timestamp,
- Communicating with other SCM repositories or tools, which means
    - Serializing[9] the files (and associated information) to be sent to collaborating[10] repositories,
    - Deserializing the files received from collaborating repositories,
    - Responding back to requests sent by collaborating repositories.

The SCM repository hosting structure must be designed in a way that the SCM repository monitoring engine can access it, transfer it and use it quickly and efficiently.

The different raw files may be human readable or not (binary format). Some are human readable (typically error files), some are not (information related to tags, labels, and deltas (differences) between source code files versions, typically compressed files[11]).

If the different files composing the last repository source code file version (and the configuration files related to the development IDE) are available in a human readable way, the repository is said to be a *non-bared repository* (you can use those files directly and "inject" them into your IDE folders).

If this is not the case, the repository is called a *bared repository* (the raw source code files can be obtained only using the SCM repository engine).

---

[9] By « serializing » we mean transforming a files structure (typically a tree) into a suite (a linear queue) that can be transferred via a typical bunch of bytes through a protocol communication pathway.

[10] We'll expand on this word later in this chapter.

[11] As a matter of fact, if this information must be transferred through a communication protocol, it must be compact.

One may think that the raw source code files are usually only code files in a programming language.

This is not true.

Those files can be Jason or XML files, for example.

Those files can also be pictures or drawings.

Those files can be MS Word files, which are considered by most SCM tools as binary files, since it is only MS Word that can handle them.

And finally, those files can be binary files such as executable files. And sometimes very, very big and large files.

The SCM tool should be able to handle all this.

Our new feature and misconception scenario.

The scenario physical infrastructure is described in the next page.

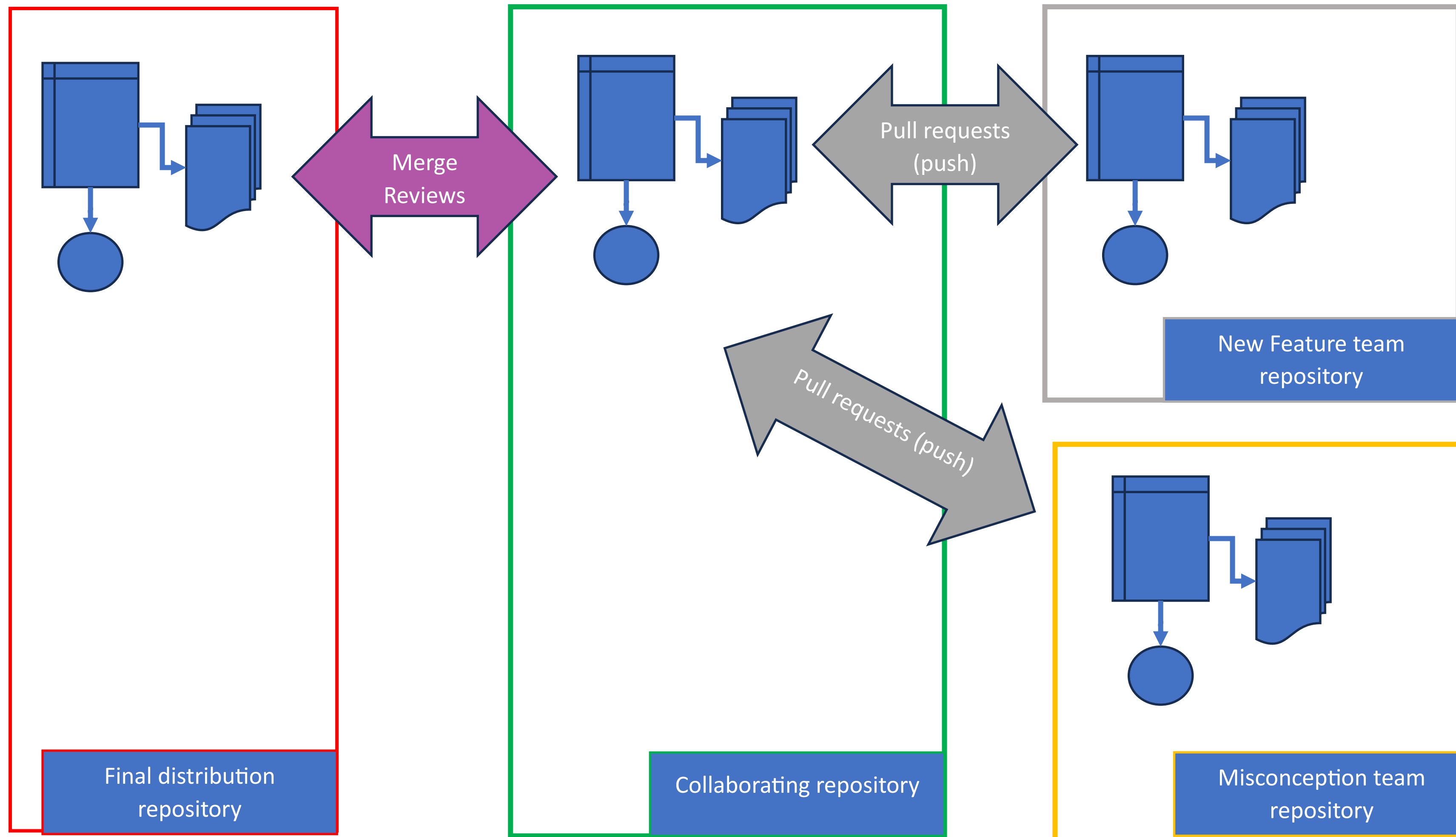


Figure 9: Feature and misconception branch scenario A.

This picture is three-folded:

- On the left side you see the repository that is used for distributing the package to the customers. Around that you should imagine that there is a complete environment mainly composed of:
    - A complete IDE, where reside the source code files and a build way for testing, debugging, and releasing the final executables and dll files (as well as needed config files). This way can be manual or automated (through the way of using nmake files for example).
    - A helpdesk teams. Ideally tickets should be handled by using some issues management provided by the SCM tool.
    - A training team dedicated to providing training sessions (and maybe tutorials or user guides) to the customers,
    - Etc.
- On the middle you see a collaborating repository. It is dedicated to collaborators which are the team(s) working on the new feature development, the team dedicated to solve the misconception area, and possibly any other needed expertise teams (internal or external). This repository will be made of a master branch and any needed additional branch such as basically the new feature branch and the misconception branch. Traditionally this repository is a *fork* of the final distribution repository (simply said: a copy).
- On the right side are the different teams repositories. Those team make changes to their repositories, test them and when they think those changes are correct check (***push***) them into the collaborating repository (what we'll call later in this chapter, they *push* them to the collaborating repository). You see on the drawing, instead of a push, *Pull requests*. That is, in this scenario, what will happen. A *push* to collaborating repository will be at some point interpreted as a request for pulling the changed code from the collaborating repository to the final distribution. This request will be the starting point for a merge review. So, in this scenario, a push from the new feature or misconception team to the collaborating repository will be transitioned as a *pull request* from the collaborating repository[12] to the final distribution repository.

[12] Which is the repository where changes made by the two developers' teams are "consolidated". A kind of intermediate between the teams' repositories and the final distribution one.

A SCM Tool, summary

A Source Code Management tool is a tool for managing source code, together with change history, comments and, most of the time the communication with other SCM tools (servers or clients).

This a usually done using different changes, branches, merge processes.

As you will see also later in this chapter, those tools also include the (base) management for bug tickets, the introduction of new features and possibly many other things.

SCM tools have their own way for storing the source code (repositories) and use different communication protocols with external parties.

In the context of this book the repository management, the way for managing branches and the communication with external parties is Git, the base standard on which Gogs is based, but also supported by other SCM tools like GitHub, Azure DevOps and JIRA, amongst many others.

Git is a protocol that can be used by tools (SCM servers and clients), but which can be used also through using command lines (we'll see it later).

The next page shows what could be the result of merging the changes showed in Figure 7, assuming that there are no conflicts between changed source code files.

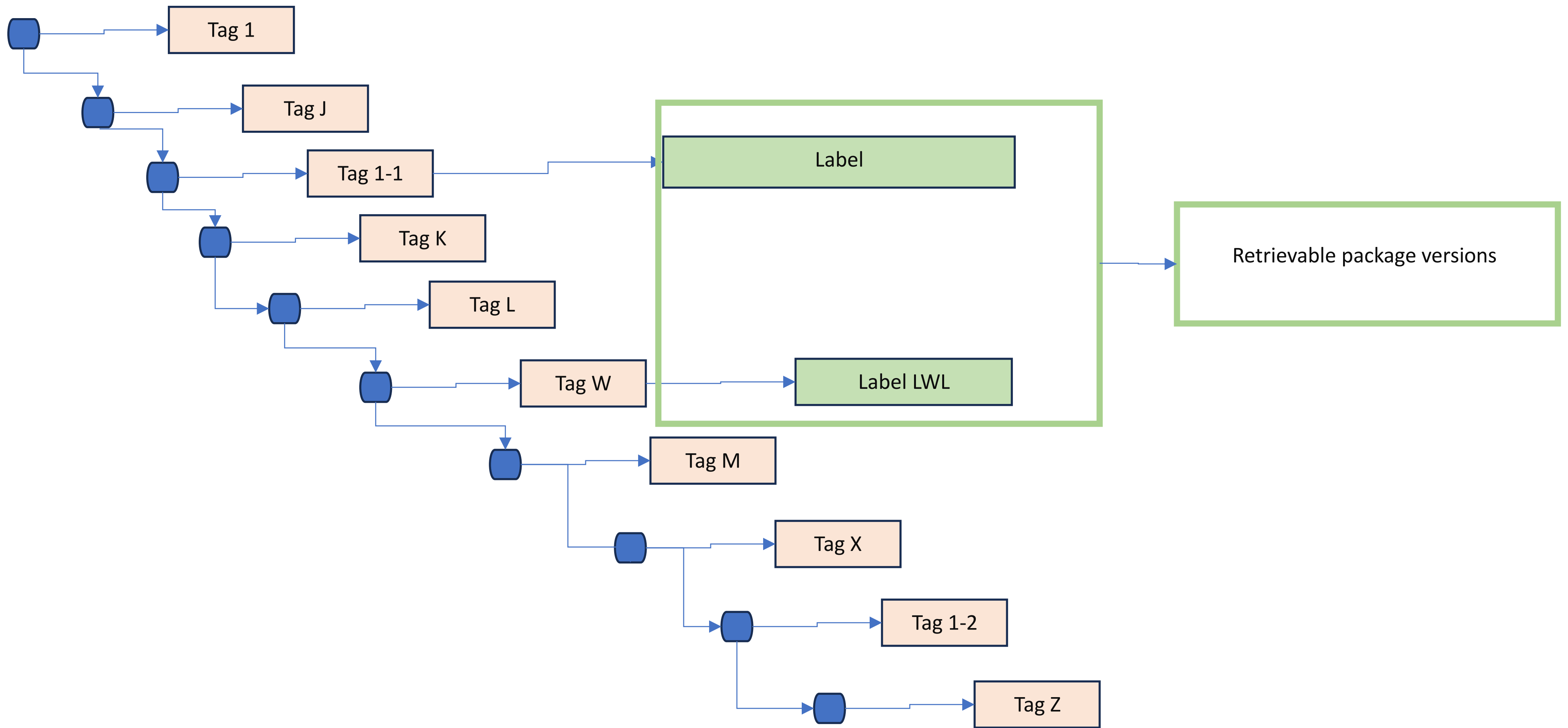


Figure 10: The final master branch.

If bugs are discovered later on, and that the Label LWL version has been installed at one particular customer premises, and that you want to install the bug fixes only at that customer premises, you can always create a branch from Label LWL and manage it separately (this could be the case if you want to avoid a large deployment from the current version you are at, for example), a branch that you don't necessarily want to merge.

Vice versa, if you want to develop a very specific feature for that customer, you can proceed the same way.

Git in a nutshell

What is a Git collaboration situation?

Well, to describe it, imagine you have a computer, let's call it the *main repository*, where resides the repository of an application package, and around the world, several different actors want to collaborate and "enrich" this main repository using their own repository, as illustrated below:

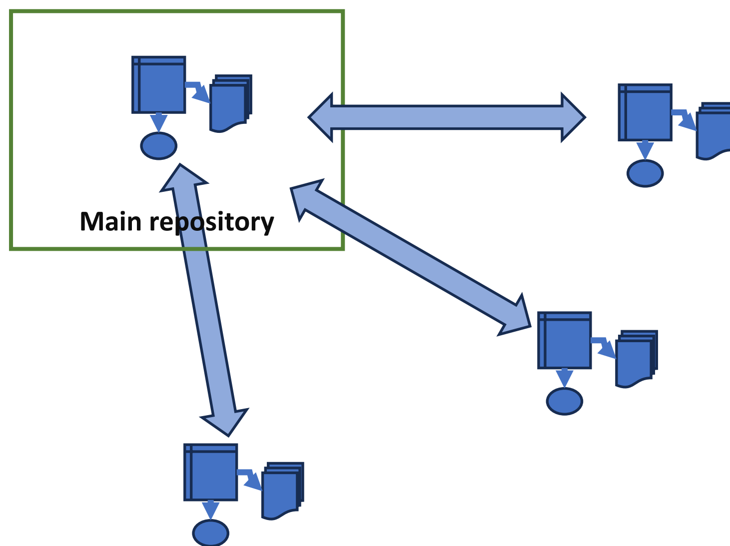


Figure 11: Git collaborating repositories.

Git provides a way for creating and managing those repositories, as well as managing the communication between them thanks to commands described in the description of the git main commands in chapter the git main commands.

Usually, rather than using just git, the real actors are applications around (using) git.

As an example, the main repository can be GitHub, or JIRA, or Azure DevOps, and, in our case, Gogs.

Those applications provide additional services such as ticket management, documentation and user guides, etc.

Collaborating repositories may be of several natures. The one we will use in this chapter is MS Visual Studio (see MS Visual Studio as a git client).

Many Gits client software offer just subsets of what is possible thanks to it. That is why many Git clients allow for launching command prompts thank to which you can invoke Git commands (Git Bash is one example).

Strictly speaking, a Git repository is a folder containing context information (such as the branch you're currently working on for example). If you want to handle this repository, you simply have to launch an adequate tool from this folder (Git Bash or Git Gui for example).

Some processes can be performed from the client side or from the server side (such as creating and merging branches).

As an example, the merge process can be organized from the client side (see the git instructions - the merge command as an example) or on the server side, like it is on Azure DevOps.[13]

Gogs is lacking from functionalities because most of the time you can connect it to clients or collaborating applications that do the job.

Another example of missing git functionalities is creating so called *pull requests* (see above).

Gogs does not completely offer this function from the server side (in some circumstances the merge process is not available). MS Visual Studio offers this functionality, only if you connect to a GitHub or an Azure DevOps repository. This is because MS Visual Studio has an additional communication layer on top of the base git one that allows the sending of pull requests directly to the server.

[13] Gogs does not allow for directly creating and deleting branches for example. This is why knowing about git commands is very useful. This is true also for MS Visual Studio as a client. It allows for creating branches but not for merging or deleting them – except if you use a command prompt (See The MS Visual Studio client.). Most Git presentations use GitHub or Jira on which the branch and merge operations are made on the server (GitHub or Jira) side.

GOGS

Gogs installation

This is described in the Gogs Installation. chapter. Since my main purpose was to use Gogs to manage my own source and connect it to the outside world, I must admit I did it in a quite quick and dirty way. This appendix will anyway give a guidance to do it in a very much proper way, especially if you want to use it in your internal network. Go also to Gogs github for more information.

Introduction

Gogs is a very interesting collaborating SCM to be used, especially because it is free and runs on Windows, once initialized, and once the web server has been launched (see the installation appendix), launch it through your browser[14]:

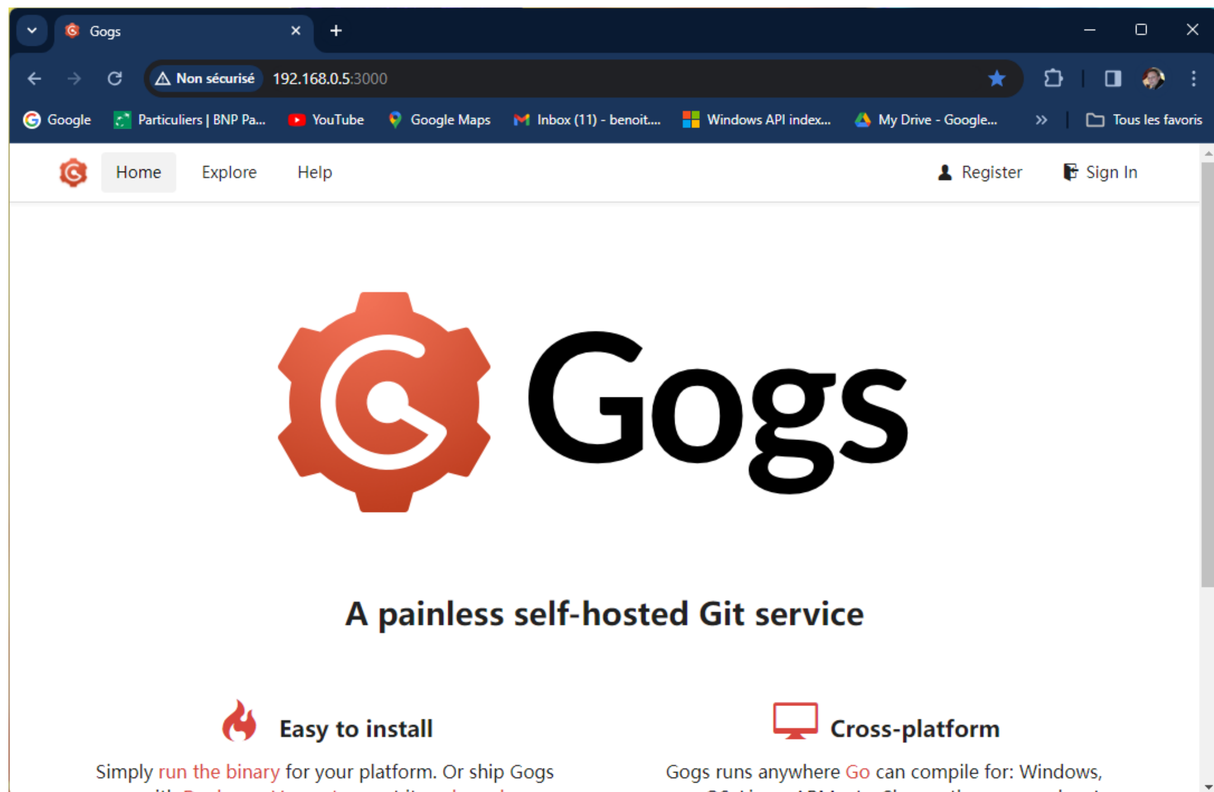


Figure 12: Launch the Gogs web page.

[14] Since we used it unsecured, you will probably have to set the adequate rights in your web browser configuration. To reach the web server page, use www.localhost:3000 as URL.

By default, first registered user (Benoit in my case) is an administrator:

Sign In

Username or email *
Benoit
Password *
Remember Me
Sign In
Forgot password?
Need an account? Sign up now.

Figure 13: Gogs sign in.

What follows is a brief overview of what you can see in Gogs. Practical scenarios will be applied in the Creating a first repository, Connecting to an existing repository, and MS Visual Studio Client chapters.

Gogs is a kind of “small” GitHub.

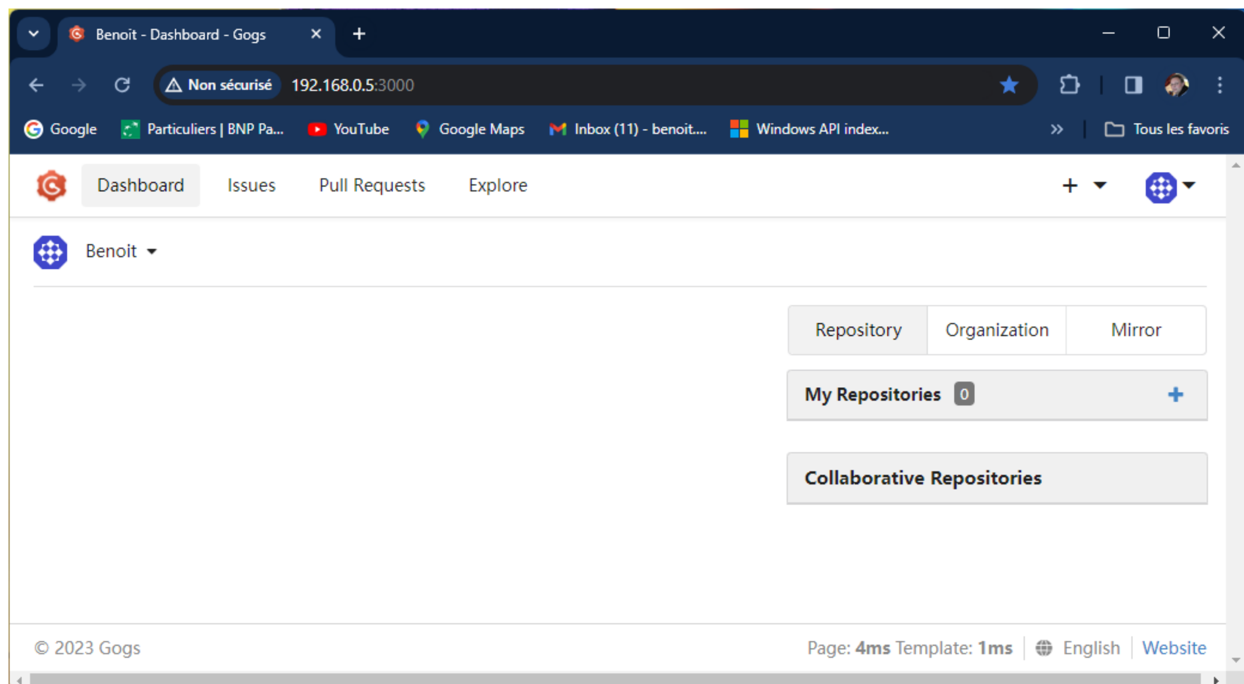
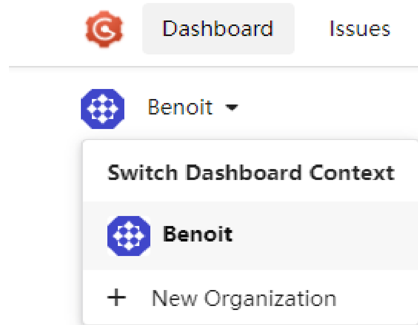


Figure 14: Gogs main functions.



The Dashboard. It contains actions assigned to you, and actions that you performed.

Figure 15: the Dashboard.

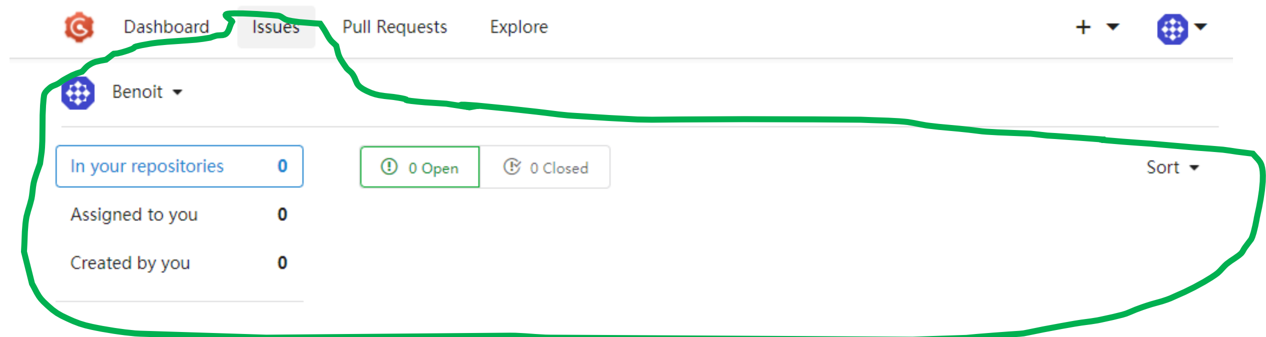


Figure 16: Gogs issues

The Issues tab: an issue is typically a bug, but also a request for a new feature, or even a question. It usually has an assignee and involve one or more participants. It allows for a discussion thread to happen:

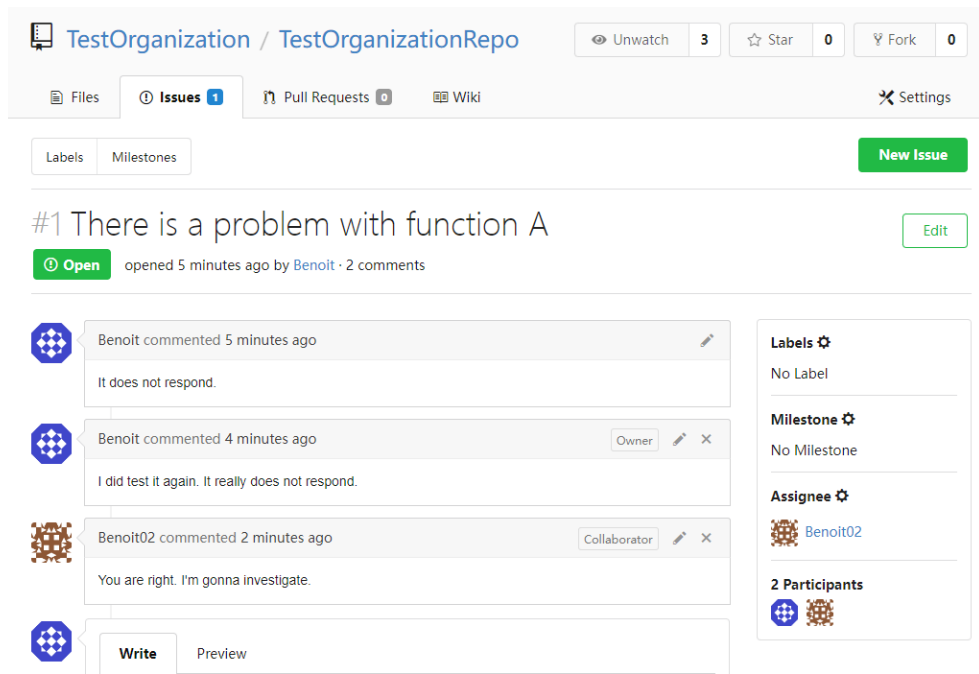


Figure 17: Discussion on an issue.

In this discussion you can see that Benoit issued a problem on function A of a package repository called TestOrganizationRepo, managed by TestOrganization, and that Benoit confirmed the function not performing correctly. Benoit02 (obviously also belonging to TestOrganization) responds that he will investigate.[15]

The Pull Requests tab:

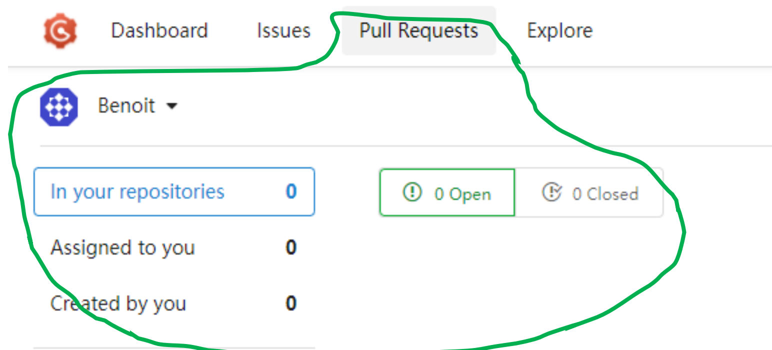


Figure 18 Pull Requests.

Pull Requests are a little bit a complex function. Gogs has a specific to treat them, as opposed to GitHub or Azure DevOps. We'll describe that later.

[15] We'll see from next pages how repositories, organizations and teams can be created and managed in Gogs.

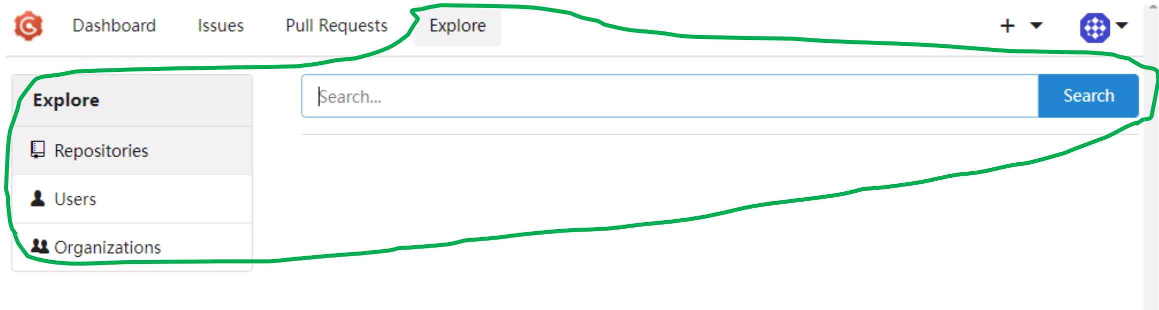


Figure 19: Exploring Gogs.

As a first step we can already draft (though the complete model goes further than that) the main different “objects” or “entities” managed by Gogs:

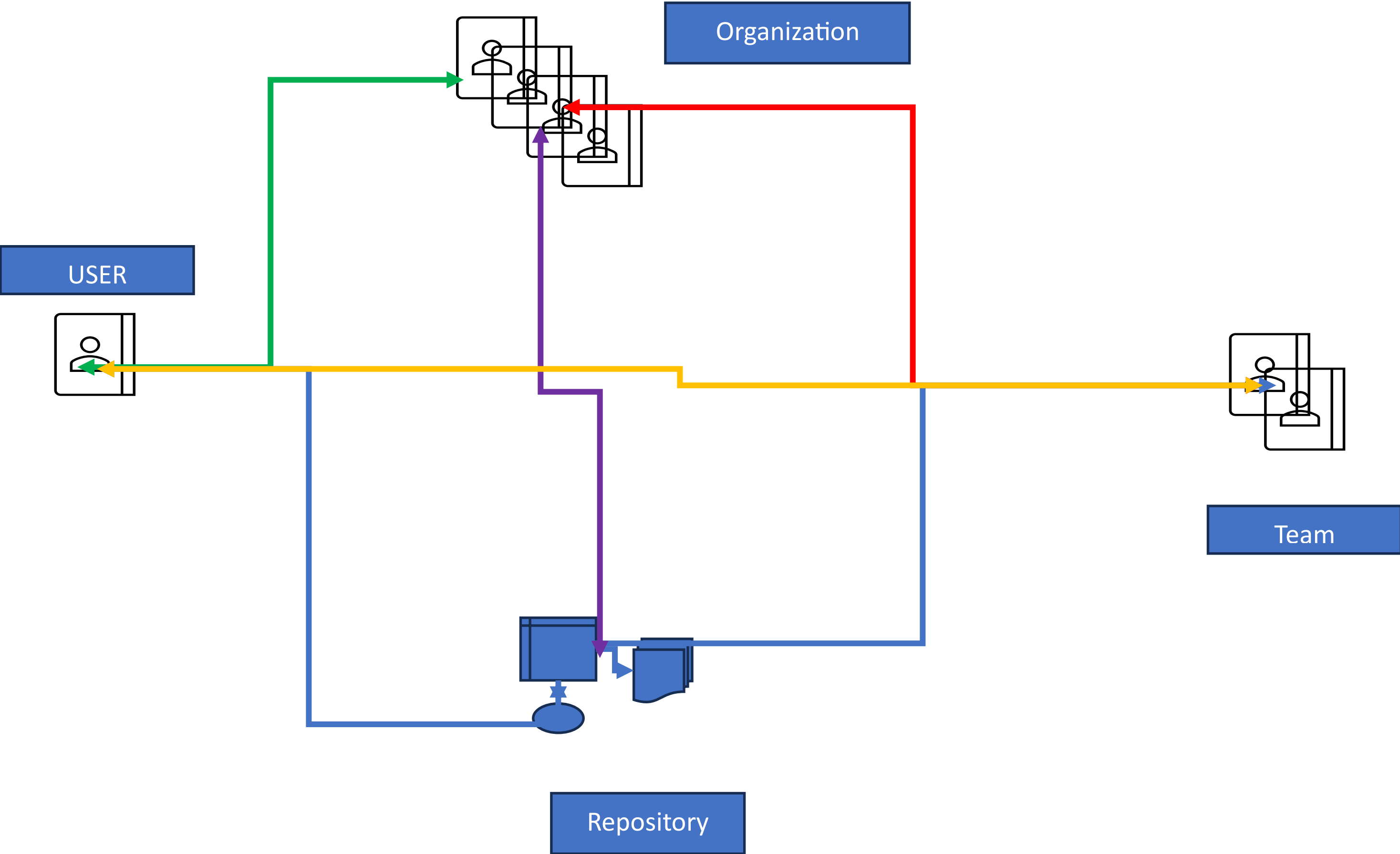


Figure 20: Users, Teams, Organizations & Repositories.

Entities & relationships.

1. A user may have different repositories (yellow arrow). When a repository is linked to a user by this only relationship, it is mainly because the repository is really private to the user, working alone with this one. Another situation is that the repository is only managed by the user, but other collaborating repositories (or organizations, or teams) are linked to it. The transfer of code from the collaborating repositories to the main one is under the only responsibility of this user. The user is administrator.
2. A team is responsible for one or multiple repositories (yellow/ blue arrow). In this team there is at least one administrator. The other users have only read access (possibility to get the source code), write access (possibility to push the code to the repository). This is a simple collaboration model (see the client/ server model described in the subchapters above).
3. A main repository is attached to an organization (purple arrow), with one or very few people. Teams a are attached to the organization, those teams are made of one or several collaborating repositories (yellow/red arrow). The teams are responsible for proposing changes through their collaborating repositories. The one or few people belonging to the main organization is (are) responsible for pushing changes to the main repository.

Continuing the tour.

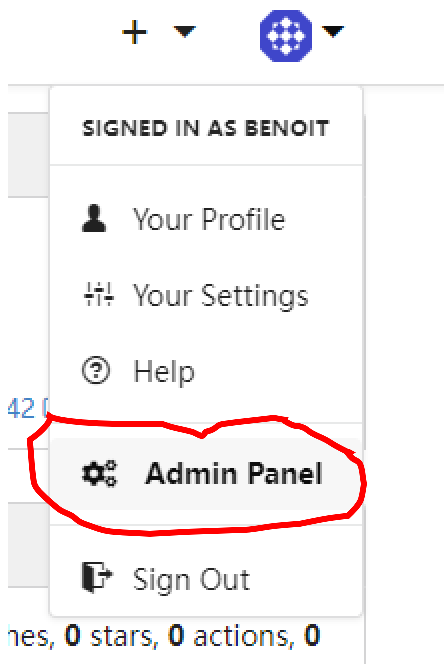


Figure 21: Accessing the admin panel.

When you are administrator, you can have access to an overall options panel:

General options panel:

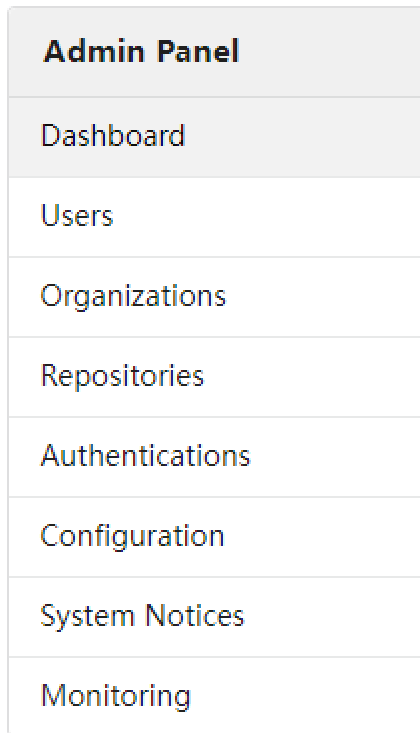


Figure 22: The admin panel.

The Admin panel is really a quick way to access to:

- A quick overall information panel with general information, statistics on the number of created accounts, repositories, organizations, etc.,
- Your dashboard,
- The user manage panel:

User Manage Panel (Total: 2) Create New Account

Search... Search

ID	Name	Email	Activated	Admin	Repos	Created	Edit
1	Benoit	benoit.borremans@gmail.com	☑	☑	0	Oct 03, 2023	✎
13	Benoit02	benoit.borremans1426@gmail...	☑	☐	0	Dec 25, 2023	✎

Figure 23: User Manage Panel

In this panel you can edit, create user accounts (useful if you want to have more than one administrator[16]).

- Organizations[17], Repositories, etc. This is something we'll discuss later in more details.

Let's come back to the general Dashboard main panel:

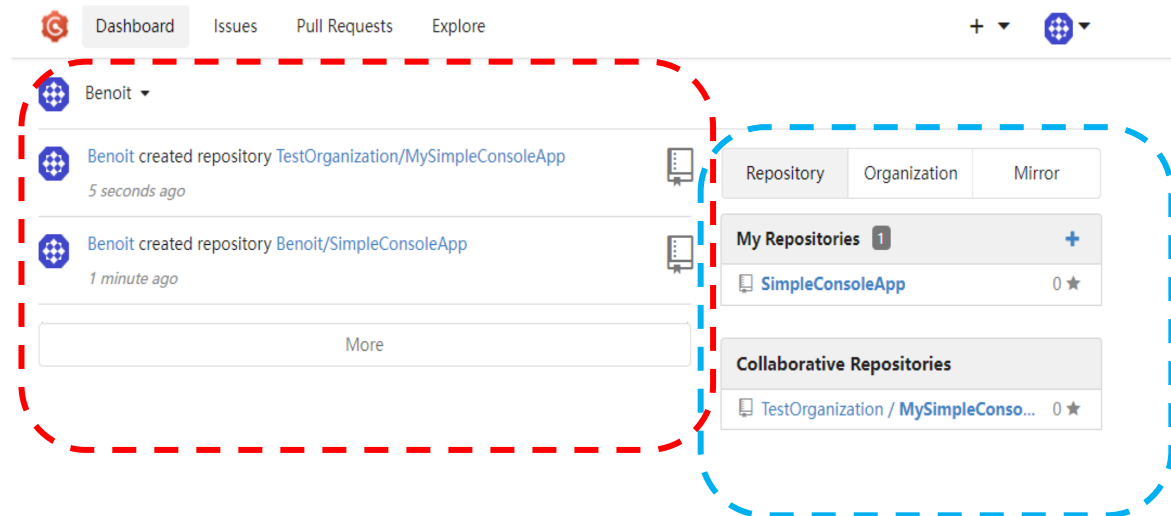


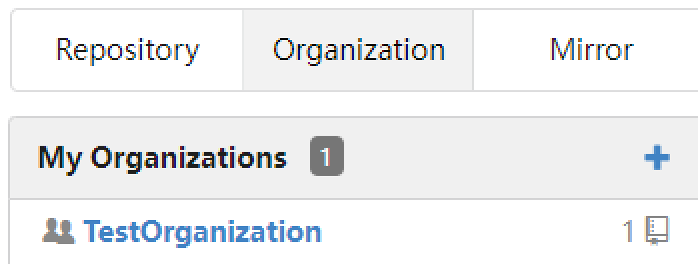
Figure 24: Main dashboard panel.

In the red/ dashed circled area, you can see all activities related to you (in this case you created two repositories).

In the blue/ dashed circled area you can see:

- Repositories that you created. The first one is a repository you created on your own name. The second one is a repository that you created on the TestOrganization organization (we'll see soon how to do this)
- Organizations:

[16] Notice that you must have at least one administrator account. Note also that this is a general panel that does not describe the rights that a user has on a particular repository (read, write, etc.). This is something we'll talk about later.
[17] That is actually the only place where you (as an administrator of course) can create an organization.



○

Figure 25: Organization sub-panel.

I you click on TestOrganization, you'll see the following interesting panel:

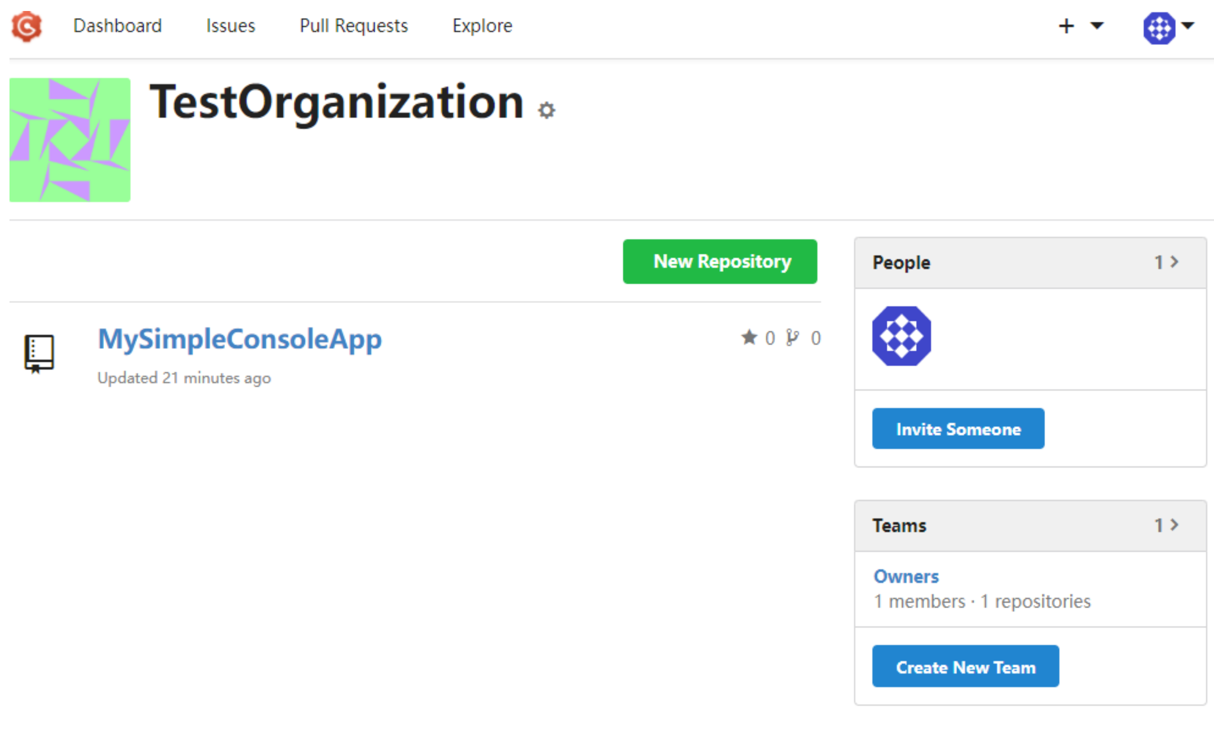


Figure 26: Organization Panel.

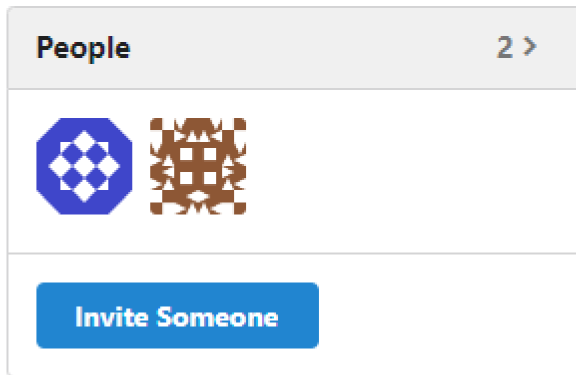
In this panel you see the following:

- You can create repositories attached to an organization,
- You can invite someone to the organization[18]
- You can create teams belonging to the organization.

[18] Be careful that inviting someone to an organization usually gives him limited possibilities to interact with the repositories belonging to the organization. It is basically eventually to create issues (without possibilities to assign someone to the issue).

You can see that one team is associated with TestOrganization (Owners). This team is created implicitly when you create the organization.

You can invite someone to the organization (in this case I invited Benoit02):



See the second avatar, corresponding to Benoit02.

Figure 27: Invite someone to an organization.

But if click on the Owners team:

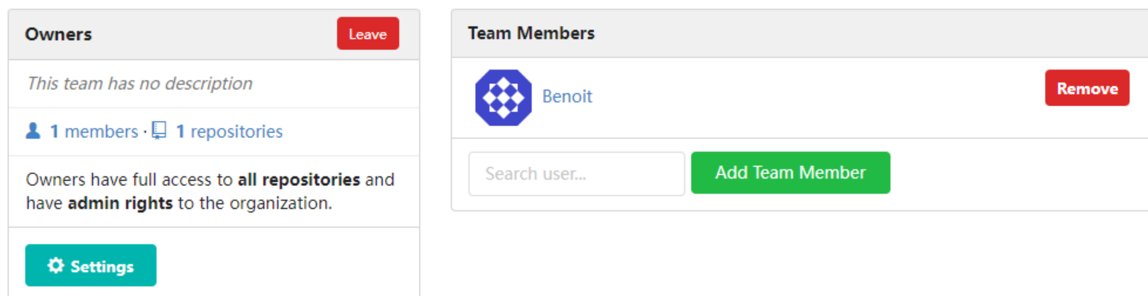


Figure 28: Organization Owners.

You see three things:

- Benoit02 is not part of the team,
- You have full access to all organization repositories and,
- you have administrator rights to the organization (meaning you can add someone, invite someone, delete repositories, etc.).

Consequently, rather than to add additional members directly to the Owners, create additional teams.

Creating a Gogs repository.

Well, it seems now that we should introduce what *should*[19] be the easiest point to understand in Gogs (and Git in general).

We already saw what a branch is, as well as (at least partly) is the merging process.

The reality is that it is pretty rare that you have to deal with a single and alone repository at a time.

Usually, you are actually working on a branch (remember, be it a new feature, a misconception, etc.) of a tree.

And the master branch of the tree is, of course, the master branch of your (local) repository.

The real (future) master branch of the package you are working on is in fact a combination of many branches from many (other) collaborating repositories. This, in fact, was already described with the new feature and misconception branches described before, considering that there is probably one master branch, one new feature branch and one misconception branch stored centrally on a server … but actually many (altered) copies of them in many different local repositories.

Anyway, at any moment in time, there must be only one repository that is considered to be the one from which official versions of your package are delivered. The situation is then something like:

[19] You will see it is indeed not the case!

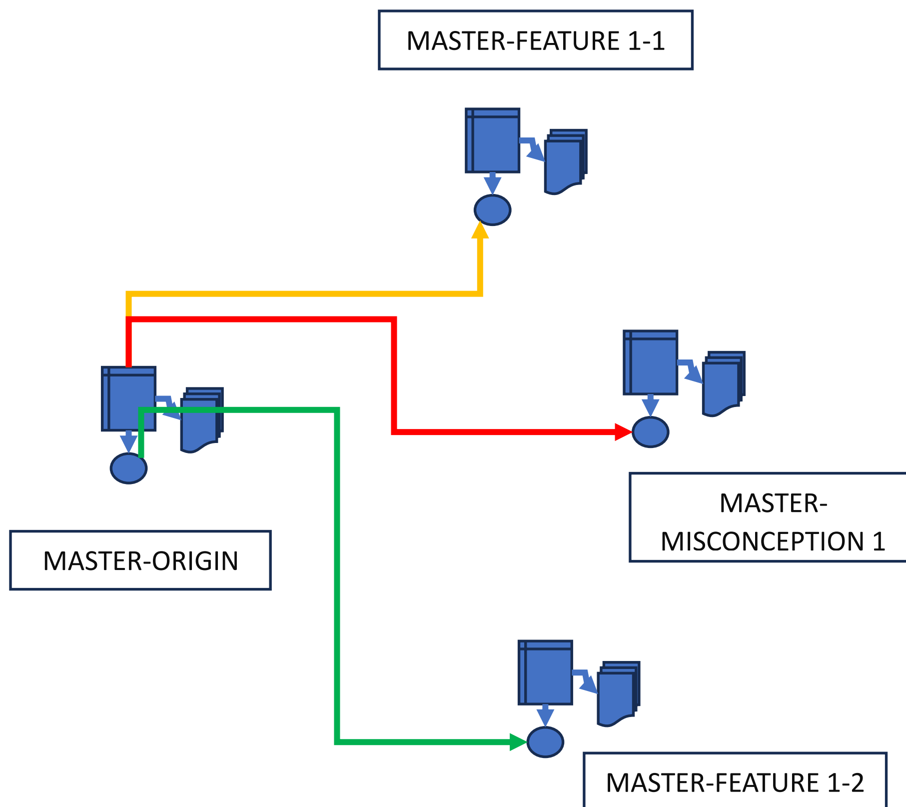


Figure 29: Different collaborating repositories.

You then have:

1. A Master-Origin repository, the repository used for generating the official version,
2. A Master-Feature1-1, the part of the new feature developed by a developer or a (team) group of (yellow arrow)
3. A Master-Misconception1, the part of the misconception developed by a developer or a (team) group of (red arrow)
4. A Master-Feature1-2, the part of the new feature developed by another developer or a (team) group of (green arrow)

All of those repositories have the same *origin*. Physically it is either a web (URL) or an SSH address.

All the different colored collaborating repositories are copies of the Master-Origin repository. They are “positioned” on their respective branch (feature or misconception).

N.B.

Actually, a Git repository is a folder in which there is a “.git” sub-folder.

This subfolder contains, amongst others:

- *The information telling on which branch you are located,*
- *The whole repository history (in the form of binary files such as tags and deltas between the different file versions),*
- *Log files,*
- *Etc.*

If the folder is an MS Visual Studio one, it contains also a ".vs" sub-folder.

If the folder contains a non-bare repository, a version of the actual package (source code) file are also present. Those files are the ones that belongs to the branch you're currently on[20].

This way, tools like Git Bash or Git Gui, when launched from the folder, know about what branch is the current one[21].

The Master-Origin has also two branches for the new feature and the misconception. When new code files have to be transferred to the Master-Origin, it has to be transferred on the correct branch (origin-feature or origin-misconception).

[20] If you invoke the switch command, or if you go to another branch in the MS Visual Studio, then these files will be replaced by the one belonging to the branch you are switching to.

[21] Gogs repositories are folders named "RepositoryName.git". Those are bare repositories and you cannot handle them by launching tools like Git Gui or Git Bash from them.

As you can understand now, the Master-Origin repository is not a repository that can be used directly for generating the official versions of the package. After new feature and misconception teams will have finished their job, the misconception and new feature branch will have to be merged to a main branch. Meanwhile, the Master-Origin repository is a *bare repository*[22].

As such, it is easily understandable that any Gogs repository is a *bare* repository. It is always used to consolidate the development of more than one developer.

Thus, in most cases, a Gogs repository will *need to be the origin of a non-bare external repository*.

Let's create our first Gogs repository.

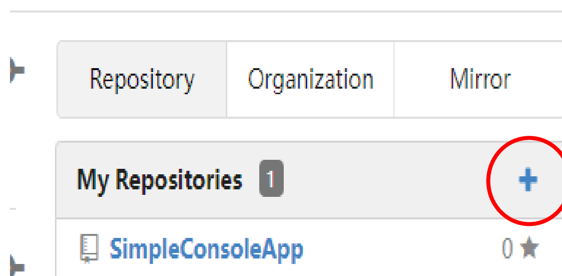


Figure 30: Create a first repository.

From the dashboard, let's create a new repository.

[22] It could be like that further on with the different collaborating repositories. If multiple developers are working on the same collaborating repository, then, again, and at its turn, this collaborating repository is a *bare* repository.

Figure 31: MyFirstRepository.

Put any description that you want. Let's focus on three important things:

1. The gitignore option (red circled), usually describe the type of files you don't consider to be part of the source code, usually files that are part of the development IDE you are using. Since we create only bare repositories in Gogs, and that we will use (later on) MS Visual Studio, we choose Visual Studio.
2. The type of license (if you publish your package code) (green circled). See Git SCM documentation about this (Git SCM).
3. For the moment, don't initialize the repository (blue circled).

You'll then obtain the following screen:

Benoit / MyFirstRepository

Unwatch 1 Star 0

Files Issues 0 Wiki Settings

Quick Guide

Clone this repository Need help cloning? Visit Help!

HTTP SSH http://localhost:3000/Benoit/MyFirstRepository.git

Create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin http://localhost:3000/Benoit/MyFirstRepository.git
git push -u origin master
```

Push an existing repository from the command line

```
git remote add origin http://localhost:3000/Benoit/MyFirstRepository.git
git push -u origin master
```

Figure 32: First uninitialized repository.

Note the HTTP address that is mentioned. This is the repository URL. You can copy it to the clipboard thanks to the button at the right of the URL[23].

As already mentioned, we will need another, local, repository to feed the Gogs repository we just created.

The first set of instructions that are listed below are the one that you should use in order to create this local repository[24].

The second set of instructions describes the instructions to be executed if you connect the Gogs repository from an existing local repository.

Those two operations are described in the chapters below.

[23] Note that you can also obtain the SSH address. Since we'll mainly access the repositories through HTTP, we won't use this option in this chapter.

[24] It is assumed, of course, that you installed a Git SCM server on your computer. Additionally, it is assumed that you installed Git bash and Git Gui tools.

Starting the game

Introduction

In the following chapters you will learn:

- How to create a new repository initially, feeding the Gogs repository from the client side,
- How to connect to an existing repository, creating branches, merge the branches and delete the branches,
- The fork process,
- Working with MS Visual Studio on the client side.

On the client side you can work with different client applications, as already mentioned above in this chapter.

Those are, for the main ones used in this chapter:

- Git Bash, which allows for creating a local Git repository through the use of Git commands (the main ones are described below), populate it and push it to a remote repository,
- Git Gui, which allows for creating a Git repository from a visual interface, as well as populating it and pushing it to a remote repository. In some cases, Git Gui is not always practical or does not allow for performing all possible Git operations, this is why, sometimes, it is useful to use it together with Git Bash,
- MS Visual Studio, which allows for almost all needed operations, at the exception of deleting and merging branches, for which the use of Git Bash is still needed.

There are many, many others.

Some client-side applications lack often from some operations (particularly on the server side) because those operations usually happen on the server side. Gogs does not provide all possible Git operations (such as merge or branch delete and creation, for example), the reason why we will use Git Bash or the MS Visual Studio client to perform those operations.

For the reasons mentioned above, the client-side applications often provide the possibility for launching Git command prompts or shell programs.

Before going to the next chapters, let's talk about the way Git operates on a local repository through the description of the main Git possible commands (operations).

<u>Git init.</u>

Launched in a folder *folderA*, the command git init creates a repository within *folderA,* basically a subfolder ".git"[25] which contains much information on the repository. We'll see this in the next command descriptions.

---

[25] Gogs stores its repositories in a folder named "RepositoryName.git". This is why it is impossible to launch a client-side application on those folders. This is mandatory to prevent such operations on the client side to happen, because Gogs store information about those repositories in its own database. This would make the whole system inconsistent.

The folderA files plus the “.git” sub-folder is typically a non-bare repository.

Git add filename.

When you start to populate a non-bare repository, you typically add files to the folderA folder. Then you start editing the file. To notice it to Git, you use the command *git add filenameA*. It puts filenameA in a so-called *staging area*, telling Git that you are working on this file (adding or modifying it[26]).

Git commit.

Git *commit -m “Commit message”* tells Git that you confirmed the modifications that you made on the last add files (you are *committing* the staged files).

Practically it transfers a compressed version of the added files, together with the commit message to the “.git” folder[27]. This is a *tag* or a *commit point*[28].

Git remote add.

Git *remote add origin URLName* or Git *remote add origin SSHName* tells Git that the repository is issued from or is linked to a remote *URLName* or *SSHName* repository (typically a GitHub, Jira … or a Gogs repository).

From this moment on, *URLName* or *SSHName* is also known as being the *origin*[29] of the repository, while *master*[30] is the name of the main current repository branch.

Git push.

The git *push -u origin master* command pushes (sends) the current repository master branch to the origin master branch. The -u option tells, amongst other things, to send bare minimum (compressed) information during the transfer[31].

The example below shows the Git Bash command prompt launched in the c:\TestGit\MyFirstRepositoryCopy folder (see first repository creation below), where you can see that the client repository is on the NewFeature branch[32]

[26] Actually, for noticing Git that you are modifying an already existing file, you use again the same *command git add*.

[27] In fact, just a delta (differences) with the previous files is added to the Git folder.

[28] Which will allow for transferring compressed information when transferring files from one repository to the other.

[29] To be used explicitly in other git commands.

[30] The git init allows for many parameters. One of them is the name you want to give to the master branch, “master” being the default value.

[31] When you push a local branch to a remote branch, there might be conflicts – for example the two branches don’t have the same history- other additional information that you might add to the -u will determine how to treat those conflicts.

[32] See the switch command below to see how to go from one branch to the other.

```
MINGW64:/c/TestGit/MyFirstRepositoryCopy
benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepositoryCopy (NewFeature)
$ |
```

Figure 33: Git Bash on NewFeature branch.

The previous git push command only pushes the current branch to the origin. A git push –all will push all branches to the origin (if a branch from the current repository does not exists in the origin, then it will be created):

```
MINGW64:/c/TestGit/MyFirstRepositoryCopy
benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepositoryCopy (master)
$ git push --all
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 369 bytes | 369.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To http://localhost:3000/Benoit/MyFirstRepository.git
 * [new branch]      NewFeature -> NewFeature

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepositoryCopy (master)
$
```

Figure 34: Git push --all pushes all branches.

Git push -u origin NewFeature would only push the current NewFeature branch to the origin.

<u>Git fetch</u>

The *git fetch –all* gets all origin branches[33] to the current repository.

<u>Remark</u>:

In this chapter, I will use fetches only on already initialized local repositories. Instead, when you want to create a local repository from an existing remote one, I suggest using the *git clone* command instead to initialize the local repository.

The *git fetch origin NewFeature* would get the remote NewFeature branch.

Most of the time what you want to do is a git pull (see next command).

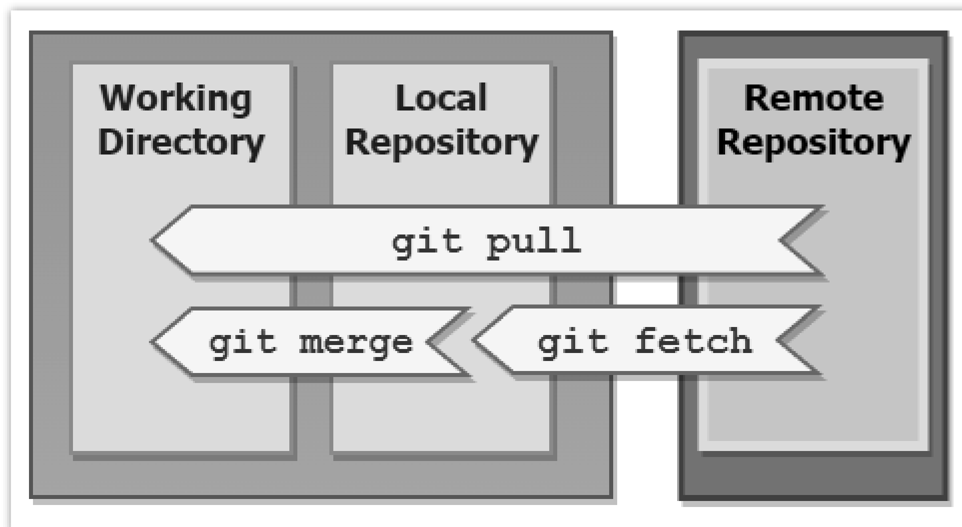
---

[33] If there are multiple origins, if you used multiple git remote add for example, it will fetch all the remotes. It could be convenient if you have different remote repositories with different branch names.

Git pull.

The *git pull --all* gets all branches from the remote repository.

The difference between git pull and git fetch is showed in the next figure:



A git pull operation is equivalent to a git fetch and merge.

Figure 35: Difference between git fetch and git pull.

See[34]

git fetch is used to retrieve the latest commits from a remote repository, but it does not create a new copy of the entire repository on your local machine. Instead, it updates your local copy of the repository's "remote-tracking branches" (branches that track the state of the remote branches), without modifying your local branches. This means that you can review the changes before merging them into your local branches.

Git switch

The *git switch BranchName* switches the Working Directory from the Local Repository branch to another one. When fetching files from the remote to the local repository, Git stores the information in the Local Repository (as showed in the previous figure). When switching from a branch to the other, Git transfers to the Working Directory (i.e. the files you're directly working with) the files belonging to the *BranchName* branch. You can see it when using Git Bash with a git switch command, it replaces the files in your Working Directory (i.e. the directory (folder) you invoked the git init or git clone command).

[34] Here you see the difference between a bare repository and a complete one. The bare repository would consist only of the Local Repository showed in the figure.

<u>Git merge</u>

The *git merge BranchName* command merge *BranchName* into the current branch.

As an example, MyFile01.txt has the following content in the NewFeature branch:

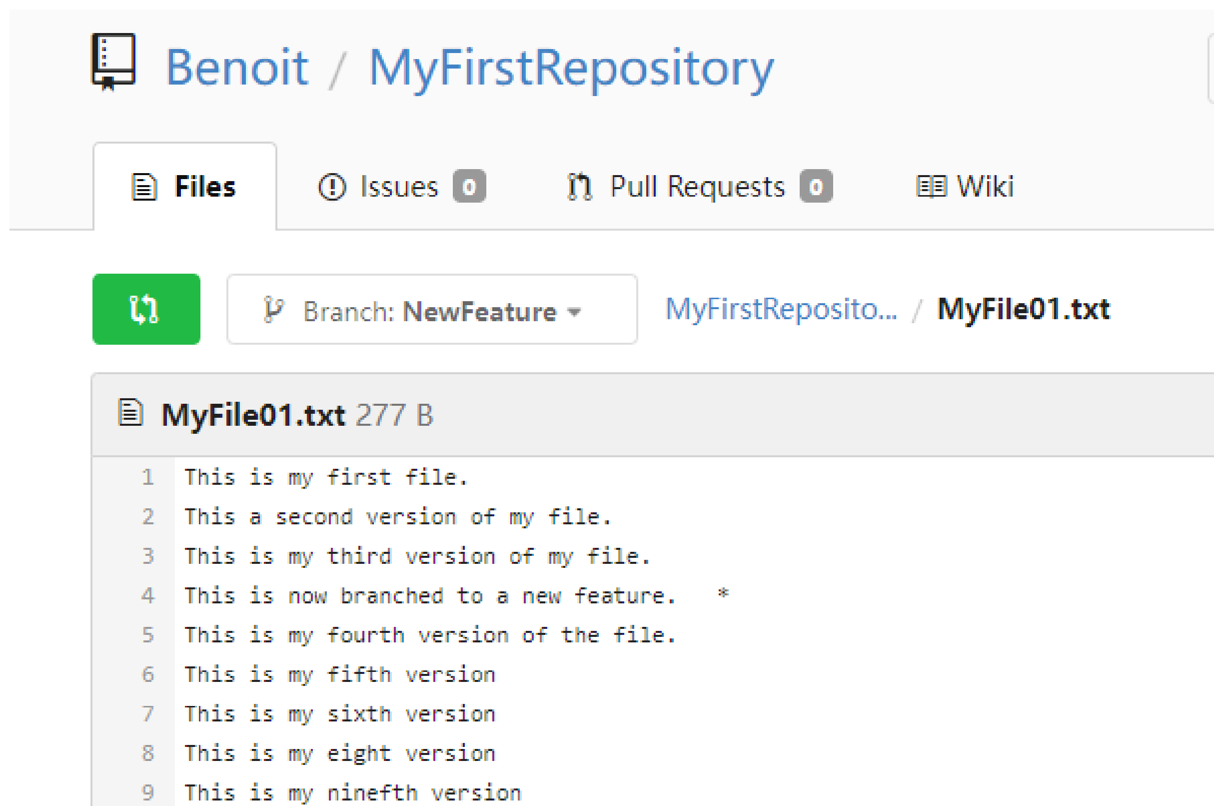


Figure 36: Content of MyFile01.txt in NewFeature branch.

And the following one in the master branch:

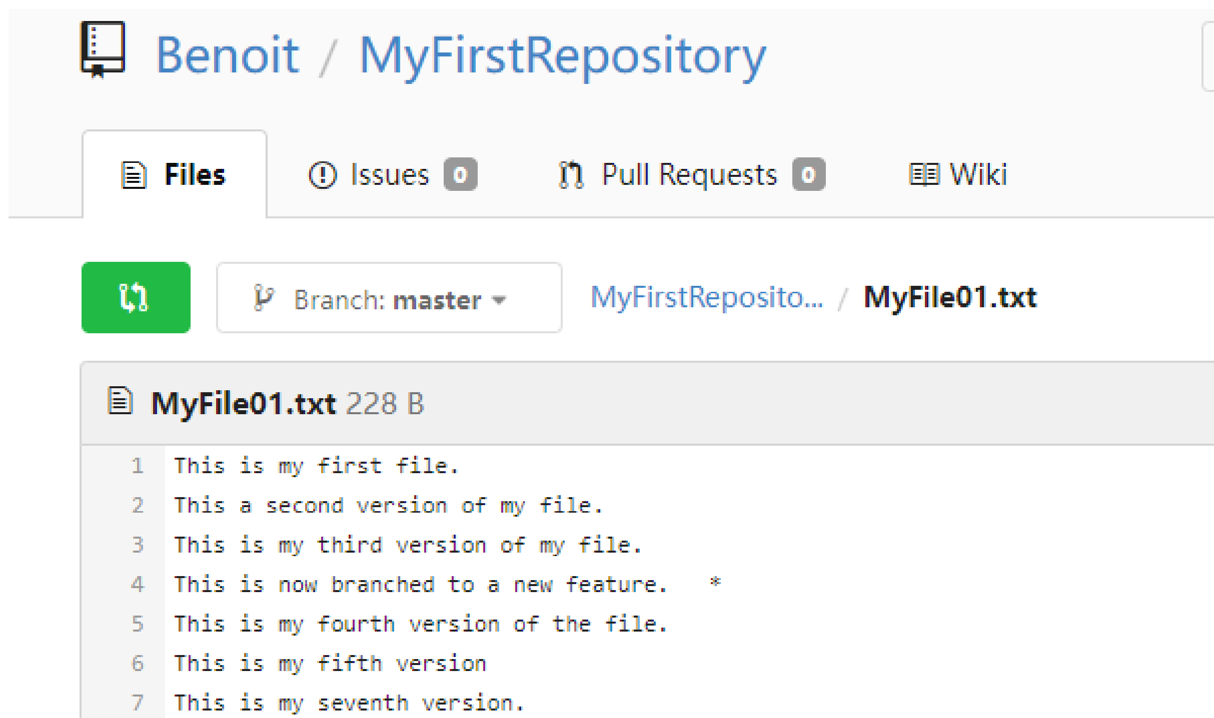


Figure 37: Content of MyFile01.txt in the master branch.

Once positioned on the NewFeature branch, the execution of a merge command would give:

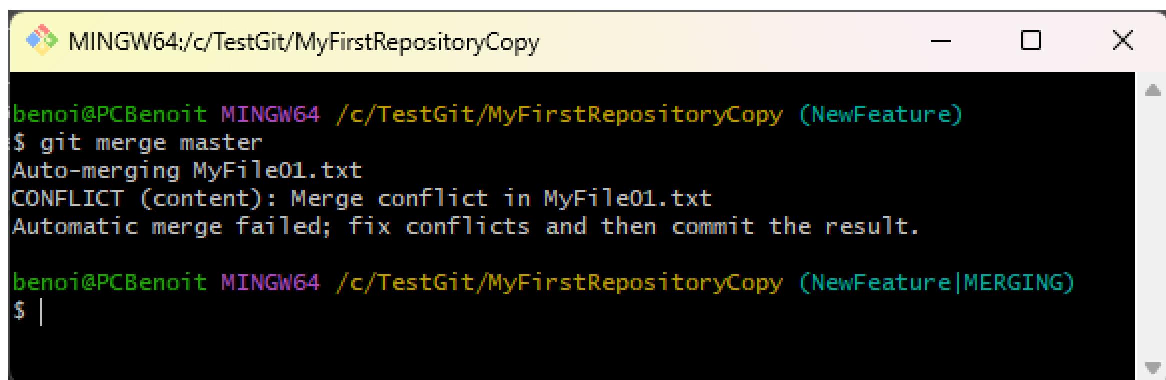


Figure 38: A merge of NewFeature in the master branch.

MyFile01.txt in the master branch and in the NewFeature branch have different histories.

If you want to go on with your working in the NewFeature branch, you have first to reconciliate the NewFeature branch with the master branch history. To do so, you:

- First consolidate the two versions in the NewFeature branch by adding MyFile01.txt into the staging area and editing it would give:

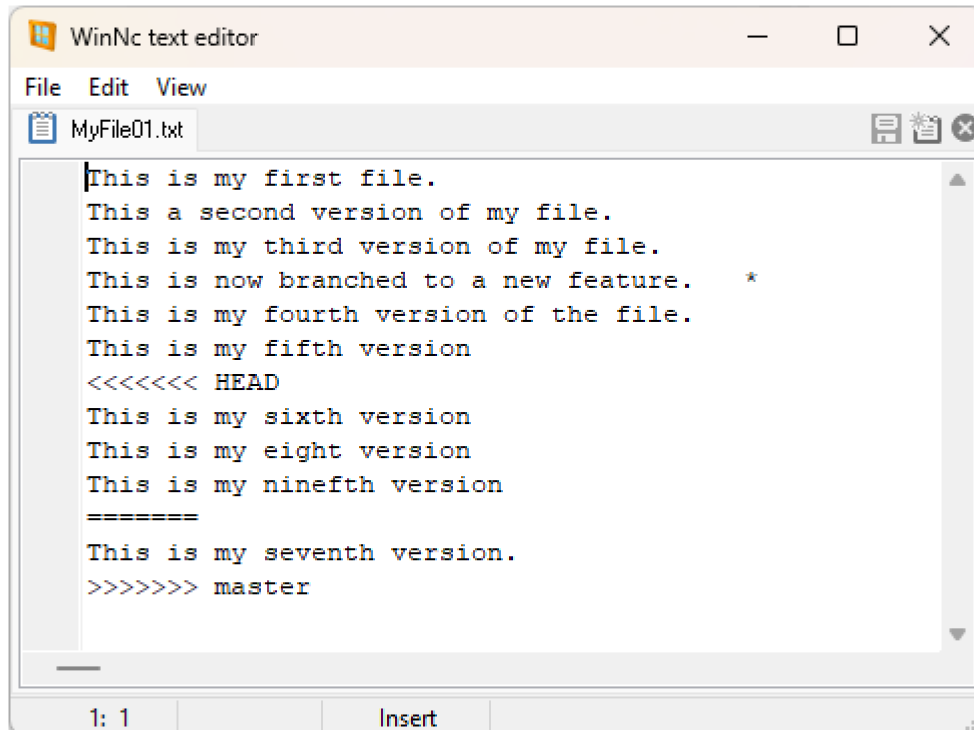


Figure 39: Editing MyFile01.txt in merge state.

And the two files reconciliated:

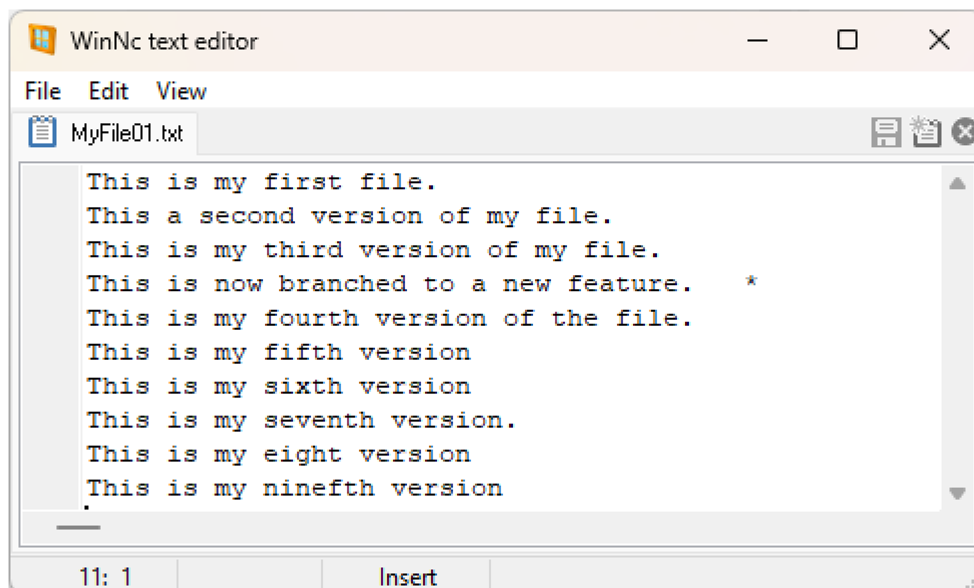


Figure 40: The two versions reconciliated.

- The complete session is:

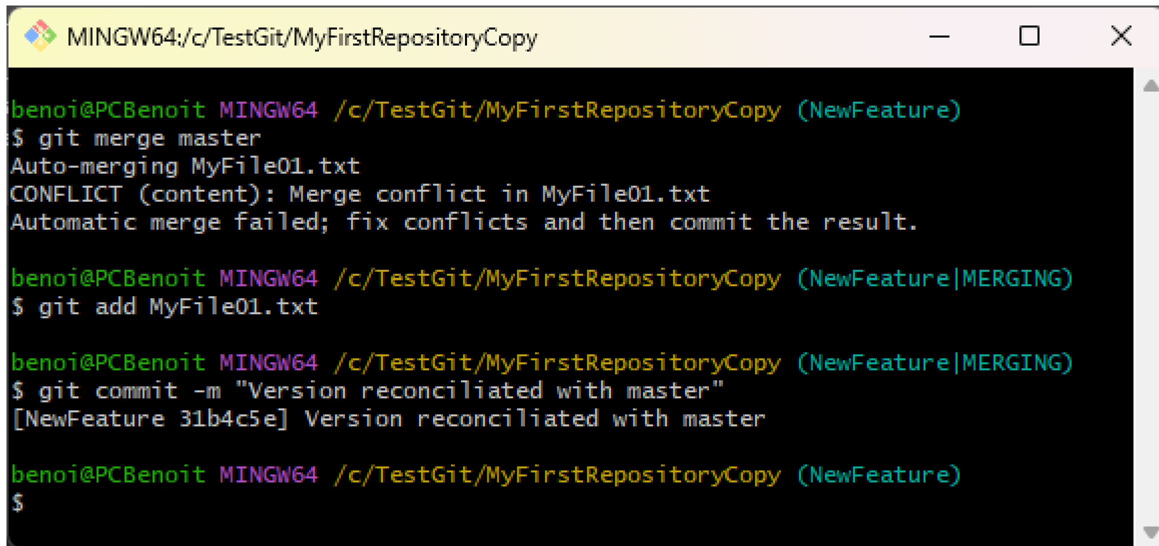


Figure 41: NewFeature reconciliated with master.

- And after pushing that all to the remote, you can then push to the remote and switch back to the master, for example:

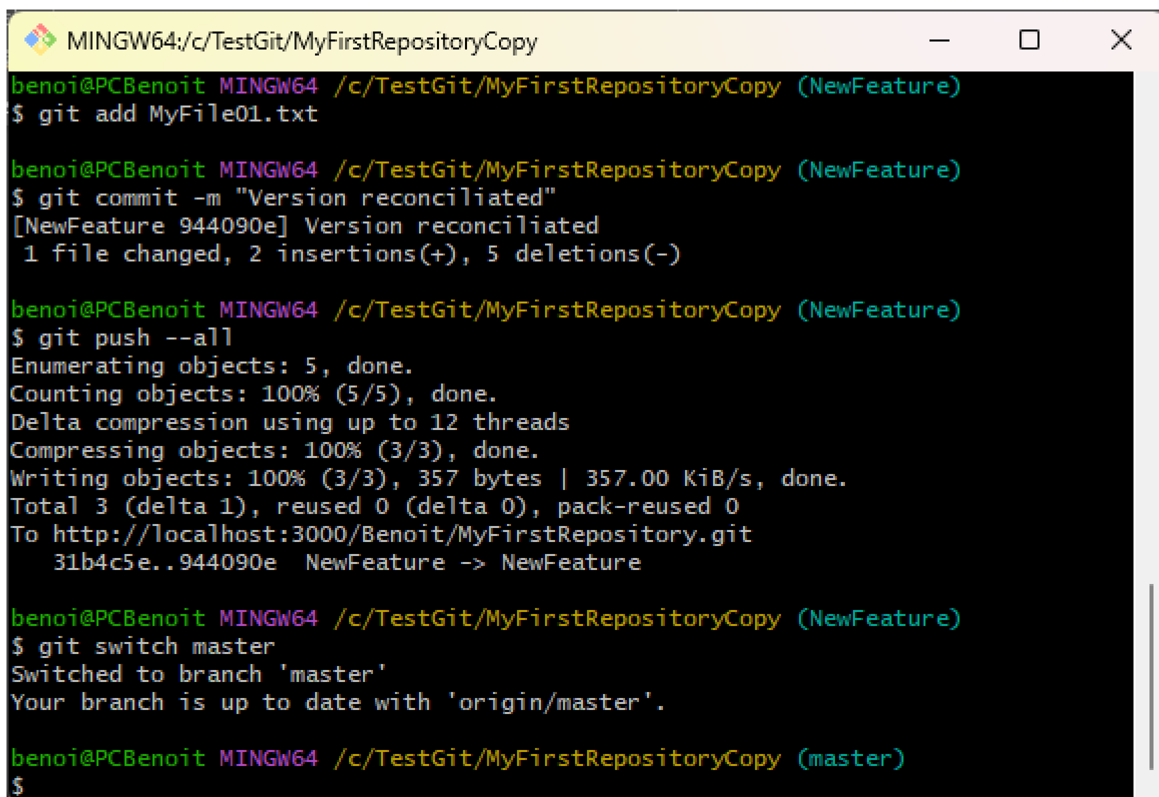


Figure 42: Back to final state.

The final picture

You can see now, considering the different git commands and their associated moves:

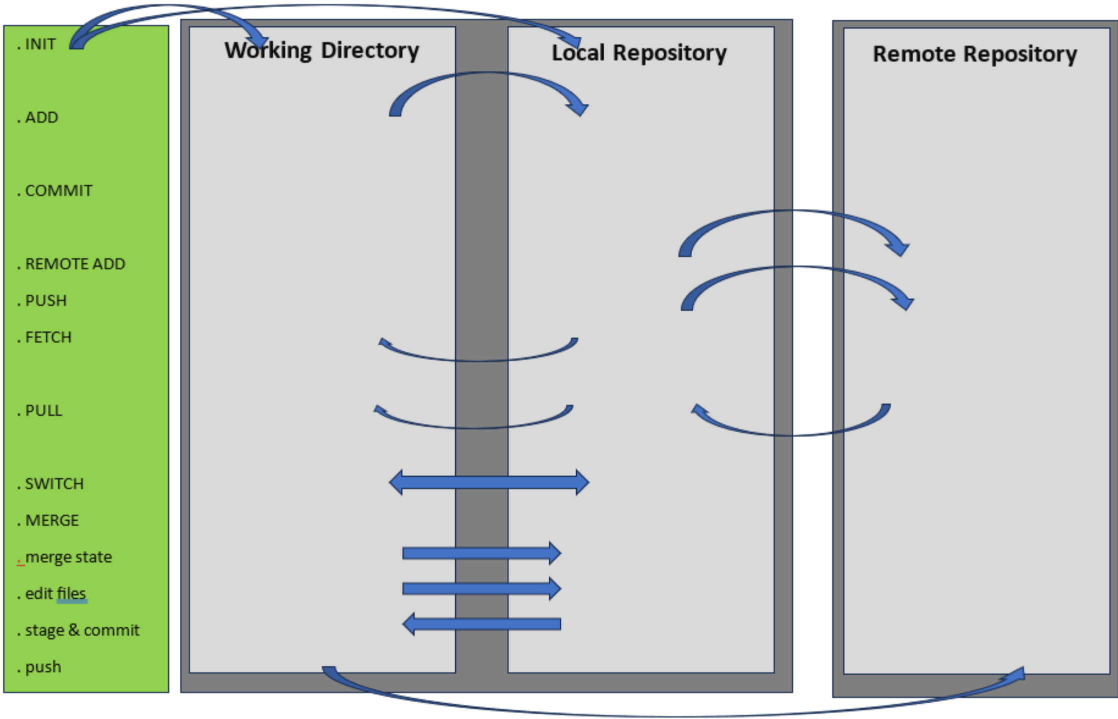


Figure 43: Git commands moves summary.

Creating a first repository

In Creating a Gogs repository. we created a MyFirstRepository repository without initializing it.

We could populate the repository by directly importing files into it. Instead, we will create an external local repository and populate the former thanks to it.

The appendix Connect the uninitialized Gogs repository to a new local one. describes:

- How to initialize a local repository,
- The addition and modifications of files,
- The pushing process,
- The creation and pushing process for a new *MyFeature* branch,
- The creation of a *pull request*.

Connecting to an existing repository

The appendix Connect the Gogs repository to an existing local one. describes:

- The launch of Git GUI and its use,
- The connection to the Gogs MyFirstRepository,
- The pull command,
- The deletion of a branch.

The fork process.

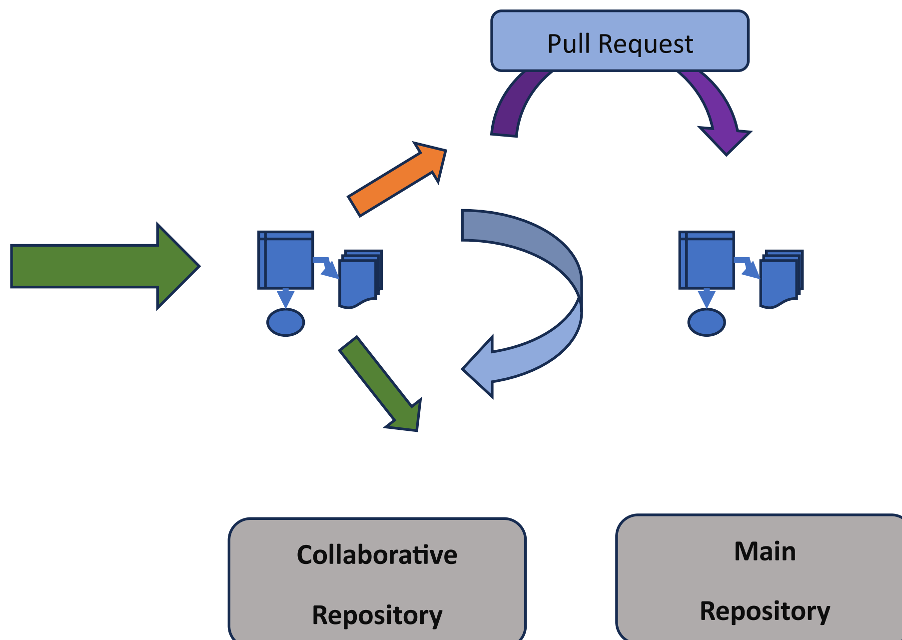


Figure 44: Pull Request from a collaboration.

The situation can be summarized as below:

- A NewFeature branch is pushed to a collaborative repository (in green),
- A comparison between the main branch (in orange) and the NewFeature branch (in green) allows for creating a pull request and assign it to somebody (in light blue),
- This pull request is reviewed, and a discussion thread can be issued, or the merge be done immediately,
- Leading to a main branch merged on the Main Repository (the comparison creates a pull request not on the collaborative repository but on the main one).

The whole and complete process is described in the appendix The fork process..

Note that you can immediately create pull requests on the main repository.

In this case you can directly clone it (rather than cloning the collaborative one) and create a new branch. But in this case, you will keep two branches on the main repository, and you'll have to delete

this branch manually from the external repository and remove the second branch, as already described previously in this chapter.

MS Visual Studio Client

In this paragraph, we will:

- Create a new repository named ASimpleApp (without initializing it),
- From the MS Visual Studio Client, create a new C# desktop application,
- Create a local Git Repository and connect it to ASimpleApp repository,
- Make some changes and push them,
- Create a branch and push it.

The whole process is described in the appendix The MS Visual Studio client..

As many other Git clients, MS Visual Studio allows for launching a command prompt session (Tools->Command Line->Developer Command Prompt). Switch to the master branch and launch the command prompt:

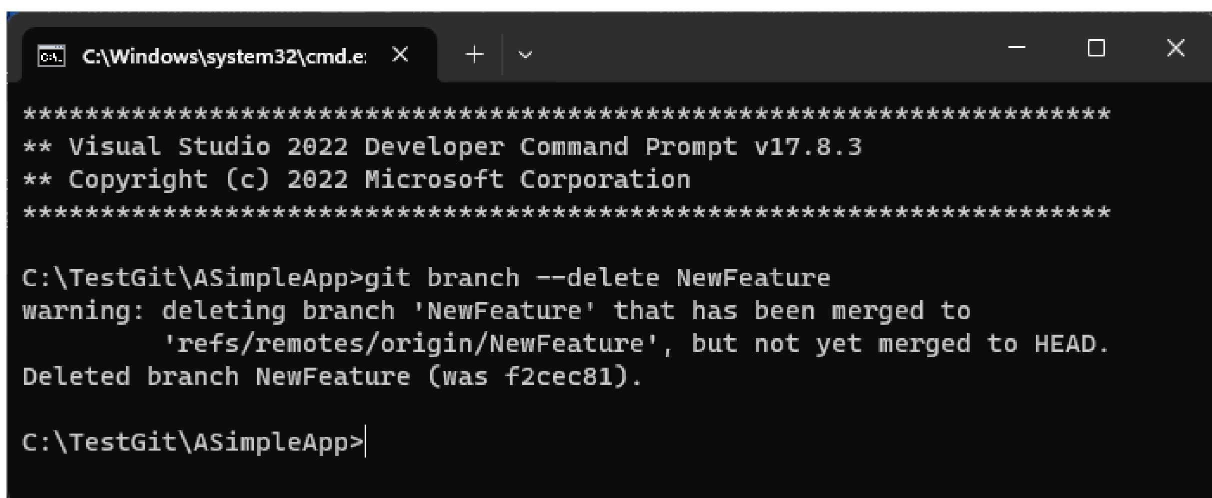


Figure 45: MS Visual Studio Command Prompt.

The NewFeature branch is now merged and deleted.

Appendices

Gogs Installation.

I did not create any particular Windows account (apart from mine, local, which is simply "benoi").

What I did is:

- Copy the whole code from Gogs code into c:\Program Files\gogs,
- Went to executables to install the executables,
- Installed SQLite3, together with a client part of it,
- Configured the installation (see what follows).

I then created a shortcut ("C:\Program Files\gogs\gogs.exe" web) to be ran into "C:\Program Files\gogs".[35]

Then you have to create a "c:\Program Files\gogs\custom\conf" folder, where you copy the file "app.ini" that you can find in "c:\Program Files\gogs\conf".

Adapt this file the following way:

- Change the RUN_USER to be your user:
    - `RUN_USER    = benoi`
- If you want the web server to a "verbose" mode [36]
    - `RUN_MODE    = dev`
- `Configure your URL, normally:`
    - `EXTERNAL_URL                     = http://localhost:3000/`[37]
- `Adapt the [database] part:`
    - `TYPE            = sqlite3`
    - `HOST            = 127.0.0.1:5432`
    - `NAME            = gogs`
    - `USER            = benoit.borremans@gmail.com`
    - `PASSWORD        = your account pa`
    - `PATH            = C:\QLiteDBs\gogs.db`[38]
- `The Gogs repositories:`
    - **`[repository]`**
    - `ROOT                           = c:/gogs-repositories`
- `Since I installed Windows in French, to force the Gogs interface to be in English:`
    - **`[i18n]`**
    - `LANGS = en-US`
    - `NAMES = English`

---

[35] That is the shortcut I launch as administrator to launch the web server. You can also install it as a running service, a choice I didn't make.
[36] Do it only when you want to debug.
[37] Notice I put it on http mode, not https. Again, a not quick and dirty way should be to let it on scripted mode.
[38] The database file where the Gogs database will be created.

There are many other options you can change. Those described here are sufficient.

Then I put all possible accesses to the user benoi on "c:\Program Files\gogs", "`c:/gogs-repositories`" and "c:\QLiteDBs" (read, write, …). I know … God bless me.

Finally, to initialize the system (mainly the database), run "./gogs web" in the "c:\Program Files\gogs" folder (use this exact spelling, included "./").

Gogs log, and error log files

The folder "c:\Program Files\gogs\log" contains especially interesting log files to look at if you have problems.

Set the option `RUN_MODE    = dev.`

Connect the uninitialized Gogs repository to a new local one.

Git SCM has been created in the UNIX world. That is why it is common to use Git commands in a Unix environment.

In Windows, you can mainly use two different tools which are emulating a Unix (or Linux) environment.

First of all, when you launch a command prompt box (be it under Unix or Windows), you must launch the tool you are using with the correct environment variables defined. That is why, when you launch a command prompt box, you have to launch it from the correct folder (directory). In Windows, this cannot be easily obtained if you launch the cmd.exe from the Windows start command and goes to the adequate folder.

That is why, using any file explorer (the Windows File Explorer included), the correct extensions must be installed in your file explorer. This can be easily verified by right-clicking into your file explorer, you can see that you can launch the desired Git tool.

As an example, for the Windows File Explorer:

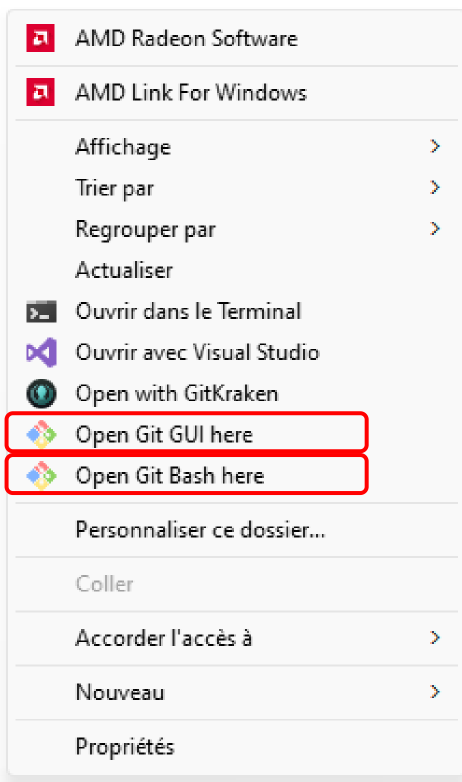


Figure 46: Open A git Client.

See the two red-circled options.

This is usually introduced in the Windows File Explorer when you download Git SCM from [Git downloads](#).

Git Bash is a command prompt that emulates a Unix command prompt and allows for running Unix shell scripts[39].

Git GUI is a more evolved console for running Git operations.

[39] Remember, Git was developed in the Unix world.

Let's create a "c:\TestGit\MyFirstRepository" folder and start Git Bash from there:

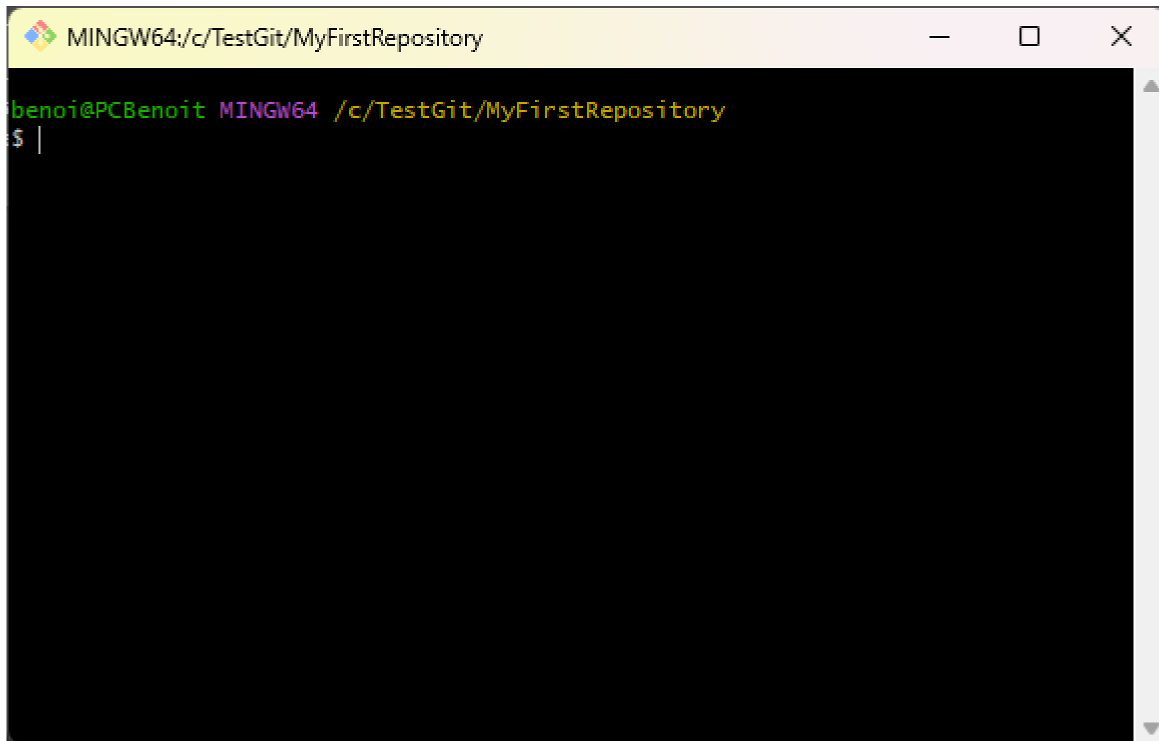


Figure 47: Git Bash Command prompt.

You see that the suggested set of instructions for creating a Git repository is:

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin http://localhost:3000/Benoit/MyFirstRepository.git
git push -u origin master
```

Let's forget about the first touch one, and let's start directly with the git init:

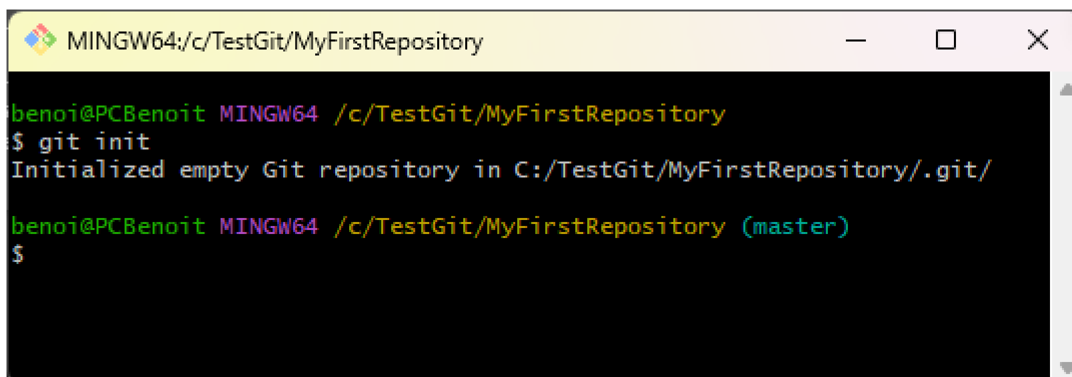


Figure 48: Git init command.

```
Initialized empty Git repository in C:/TestGit/MyFirstRepository/.git/
benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
```

You see that it tells you it created a git repository (which practically means that it added a ".git" subfolder. The second colored sentence is also interesting, since it tells you that it created a master branch on which it is "branched".

Then let's create and add a file named "README.md":

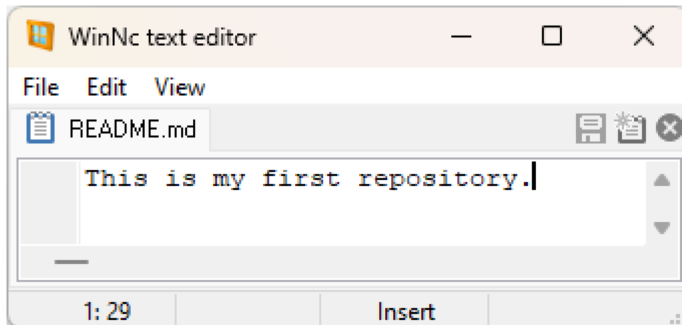


Figure 49: README file creation.[40]

The second Git instruction is to add this file (git add README.md) into the staging area (since it is not here the purpose to provide a full Git commands guide, I suggest that you go briefly to [Git SCM doc](), only main interesting git commands will be developed in this chapter).

Let's also create a second file "MyFile01.txt" into the folder:

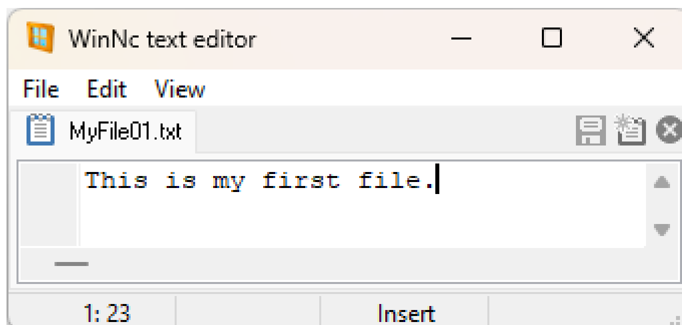


Figure 50: Create a first project file.

[40] I am personally using Windows Norton Commander (available for a very little price), which, on top of being a bi-folded file explorer, offers also a useful file comparison tool.

Let's then add those two files in the staging area:

```
MINGW64:/c/TestGit/MyFirstRepository
benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ ^C

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ git add README.md

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ git add MyFile01.txt

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ |
```

Figure 51: Adding files to the project.

The next command (git commit -m "first commit) tells the repository to commit the staged files into the repository:

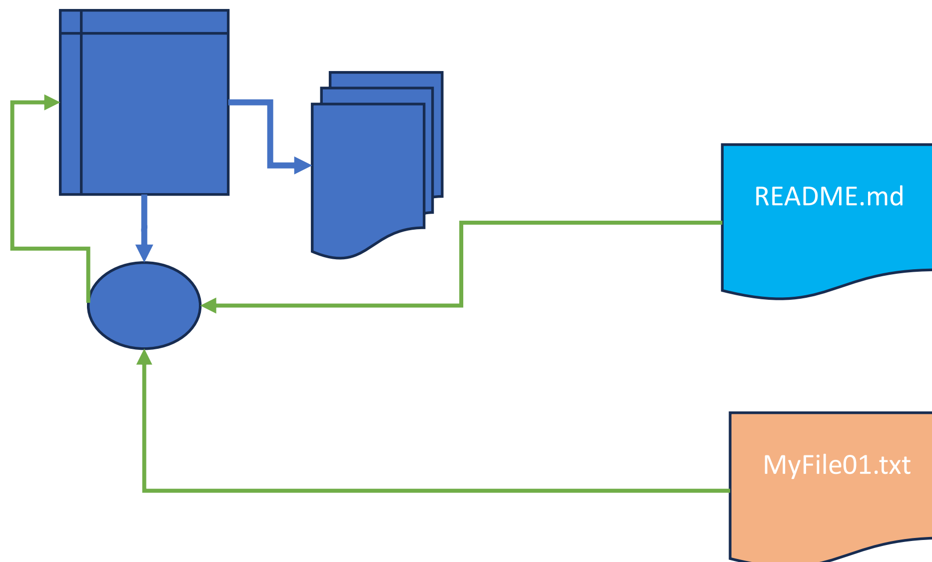


Figure 52: Feeding the repository with (a new version of) files.

The next command:

```
git remote add origin http://localhost:3000/Benoit/MyFirstRepository.git
```

tells that the current repository is linked to a remote one (our new MyFirstRepository Gogs one, the *origin*).

The next one tells to push the current master branch to the origin.

The two commands (commit and push) tell really interesting things:

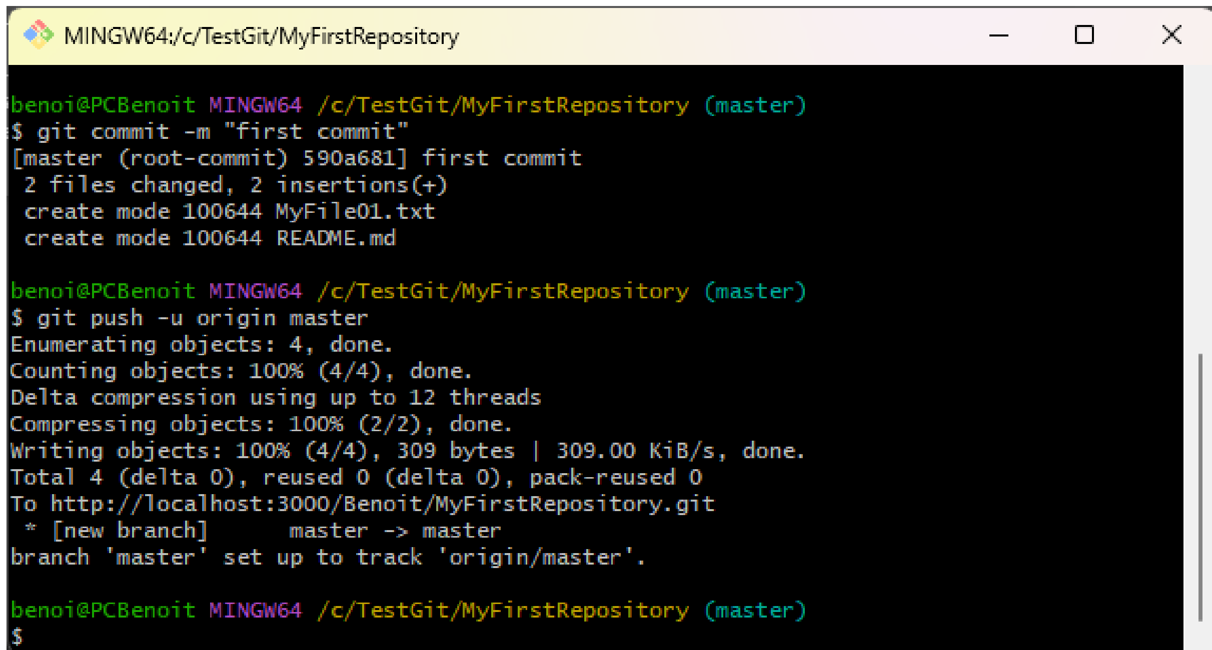


Figure 53: Git file transfer.

The git commit command tells you that you introduced a change in your package that you call “first commit”, that consists in two new or changed files and wears a *tag* “590a681” in your repository.

The git push command line tells you that it will transfer 4 objects (two of them probably be the two files, and one of them the information about the commit tag) and that it will be transmitted a “compressed” files[41].

Coming now back to the web screen on the repository (click on it to refresh it), things start now to be interesting … the game’s commencing:

[41] For more details on it, see the appendix on the git protocols and repository structure.

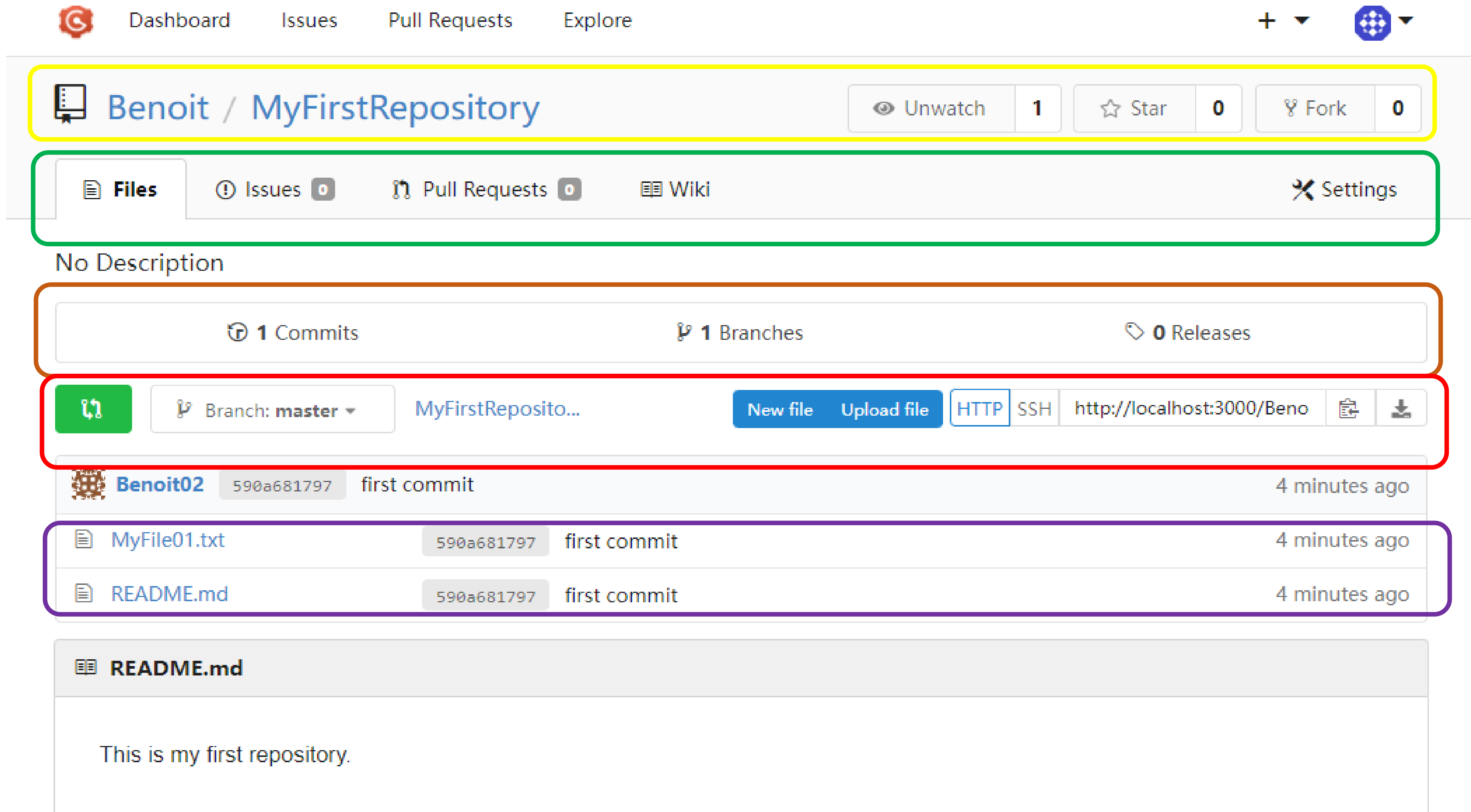


Figure 54: Gogs Repository main screen.

You can see there 5 main parts:

1. The full name repository, together with one unwatch (meaning that it is the first time you see this screen after a main action happened – the push), a starring quotation that you could for example interpret as a degree of importancy) and finally the number of forks (we'll see that point later) - Yellow.
2. The number of commits, branches, and releases – Brown.
3. The "compare" green symbol, that we'll talk about later. The branch on which the repository is positioned (or *checked out*[42]), and then a last symbol on the right side which allows to make a zip package containing the repository files. It is also possible to create or import new files directly into the repository – Red.
4. The repository files content. Note the tag associated to them (see the content below) – Purple.

If you click on tags, you see the related information:

Figure 55: Tags related information.

Notice also that now you have possibilities to create Issues and Pull Requests (the last one, again, we'll see it later).

Ok. Let's continue de game by changing files in our local repository.

[42] Check in and check out operations in Git does not represent the same thing compared to other SCM tools. In git, a repository is said to be checked out on a branch (while in other SCM, we talk about checking in or checking out files).

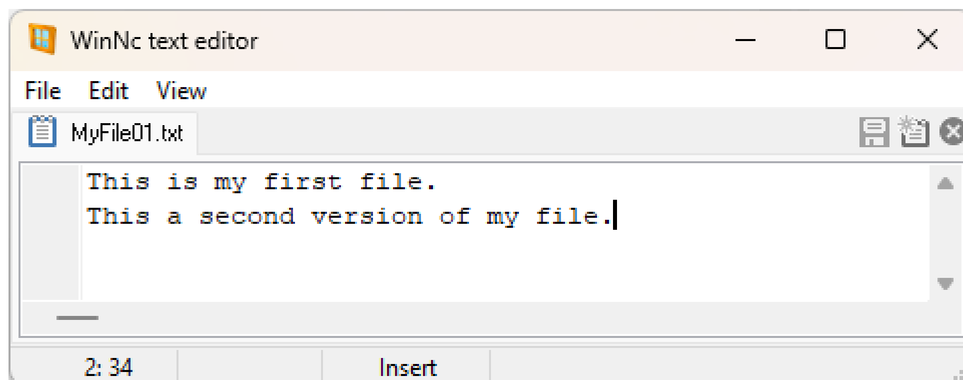


Figure 56: Changing a file.

We have now first to stage this new version of the file, then to commit it:

MINGW64:/c/TestGit/MyFirstRepository

```
benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ git add MyFile01.txt

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ git commit -m "My second version"
[master 62c1bd6] My second version
 1 file changed, 2 insertions(+), 1 deletion(-)

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$ git push -u origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 335 bytes | 167.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To http://localhost:3000/Benoit/MyFirstRepository.git
   590a681..62c1bd6  master -> master
branch 'master' set up to track 'origin/master'.

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepository (master)
$
```

Figure 57: commit and push the change.

If you refresh your browser, you will see:

Benoit02 62c1bd624d My second version			2 minutes ago
MyFile01.txt	62c1bd624d	My second version	2 minutes ago
README.md	590a681797	first commit	3 hours ago

Figure 58: pushed change.

Notice the second commit tag (62c1bd624d).

If you click on MyFile01.txt:

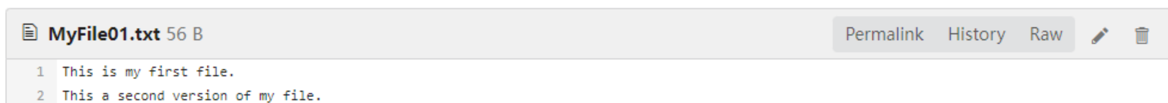


Figure 59: file changed in Gogs.

And if you click on History:

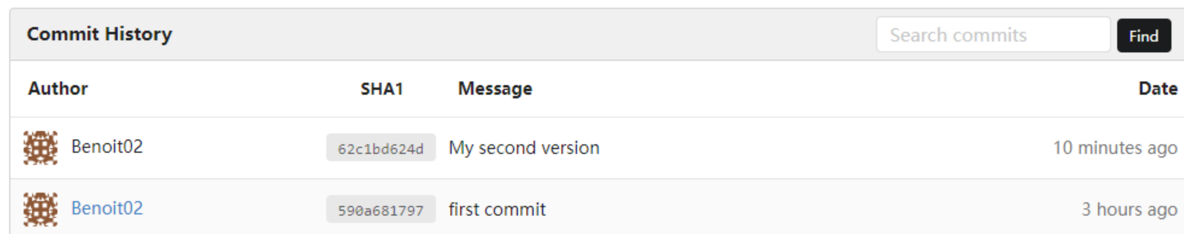


Figure 60: File History.

This shows you a bit of things you can do by directly using git commands from a command prompt.

Gits commands and their multiple arguments represent a pretty huge set of possible operations.

This is why in most of the cases, you use a dedicated Git Client GUI[43].

We'll now go on introducing Git GUI, which allows you to perform git commands without having to know the Git commands (and format!) that you should use to perform the corresponding operations.

Let's now forget about Git Bash and let's switch to Git GUI:

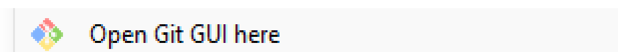


Figure 61: Launching Git GUI

[43] Those can be Git GUI, that we will still use a bit here, GitKraken, MS Visual Studio or MS Visual Studio code, etc. Those clients offer any, most of the time, a way for launching a command prompt box, since they don't necessarily offer visually a way for performing all possible operations that the whole Git SCM offer.

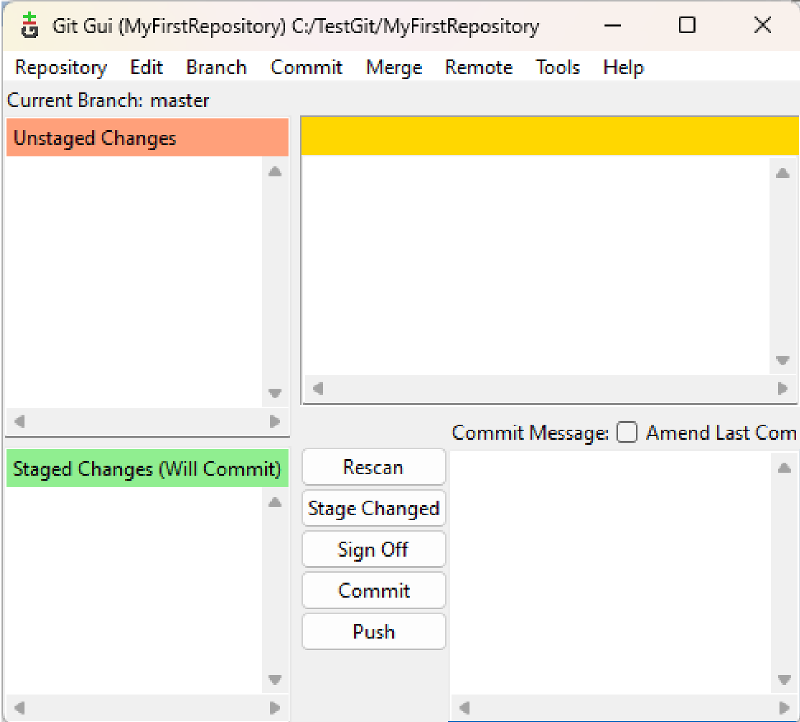


Figure 62: Git GUI.

Let’s again modify our file:

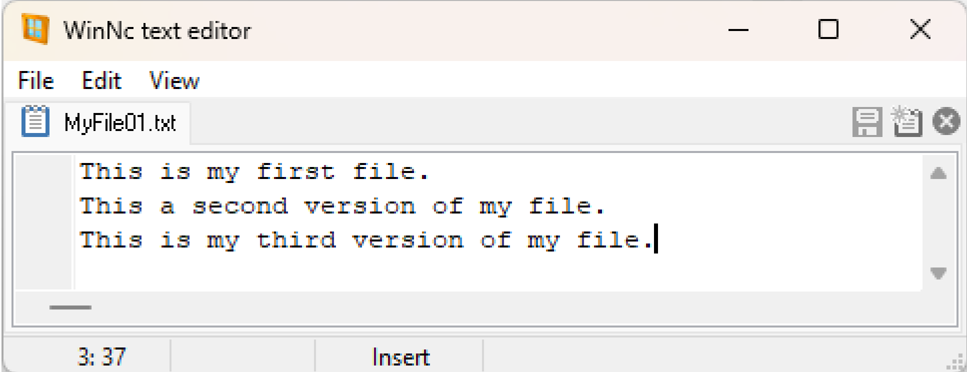


Figure 63: Third file version

You'll have to hit on the rescan button to allows Git GUI to detect you changed the file:

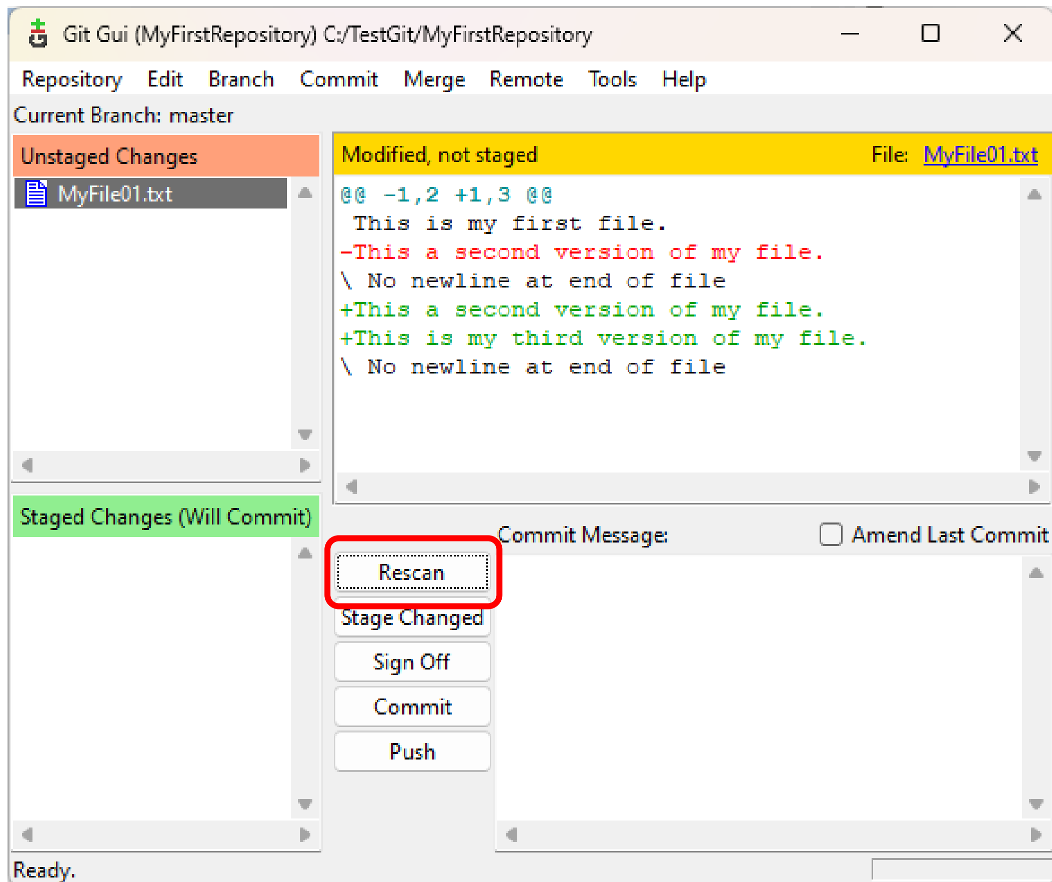


Figure 64: Changing a file via Git GUI.

Now Stage the change, add a commit message, and hit the Commit button.

Then, hit the Push button:

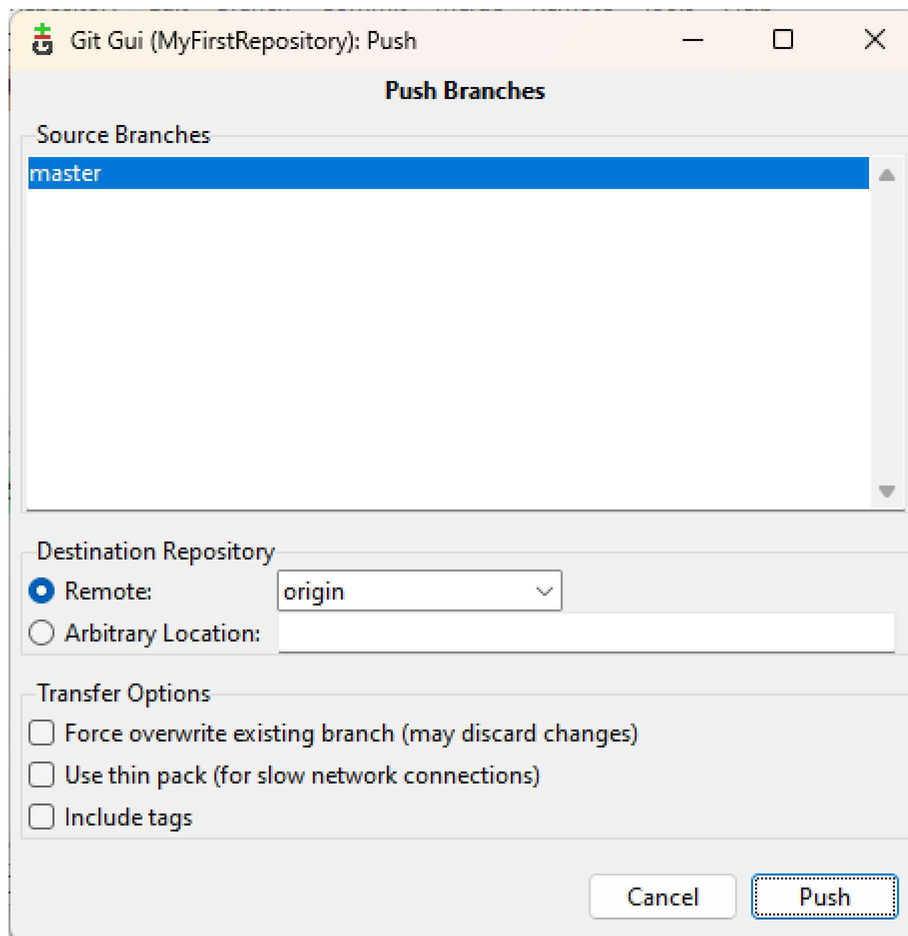


Figure 65: Git GUI - push the change.

Hit the Push button again.

You'll obtain the following popup:

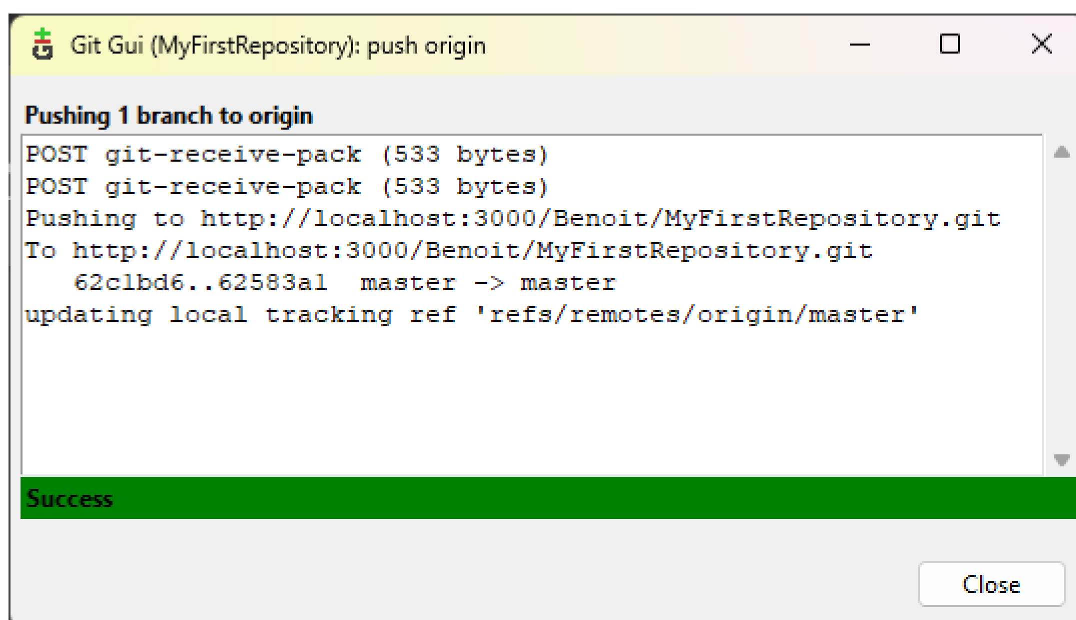


Figure 66: Push Popup.

Close it.

If you go to the Gogs GUI, you'll see the corresponding changes.

From the Git GUI, let's create a new "NewFeature" branch (Branch->Create):

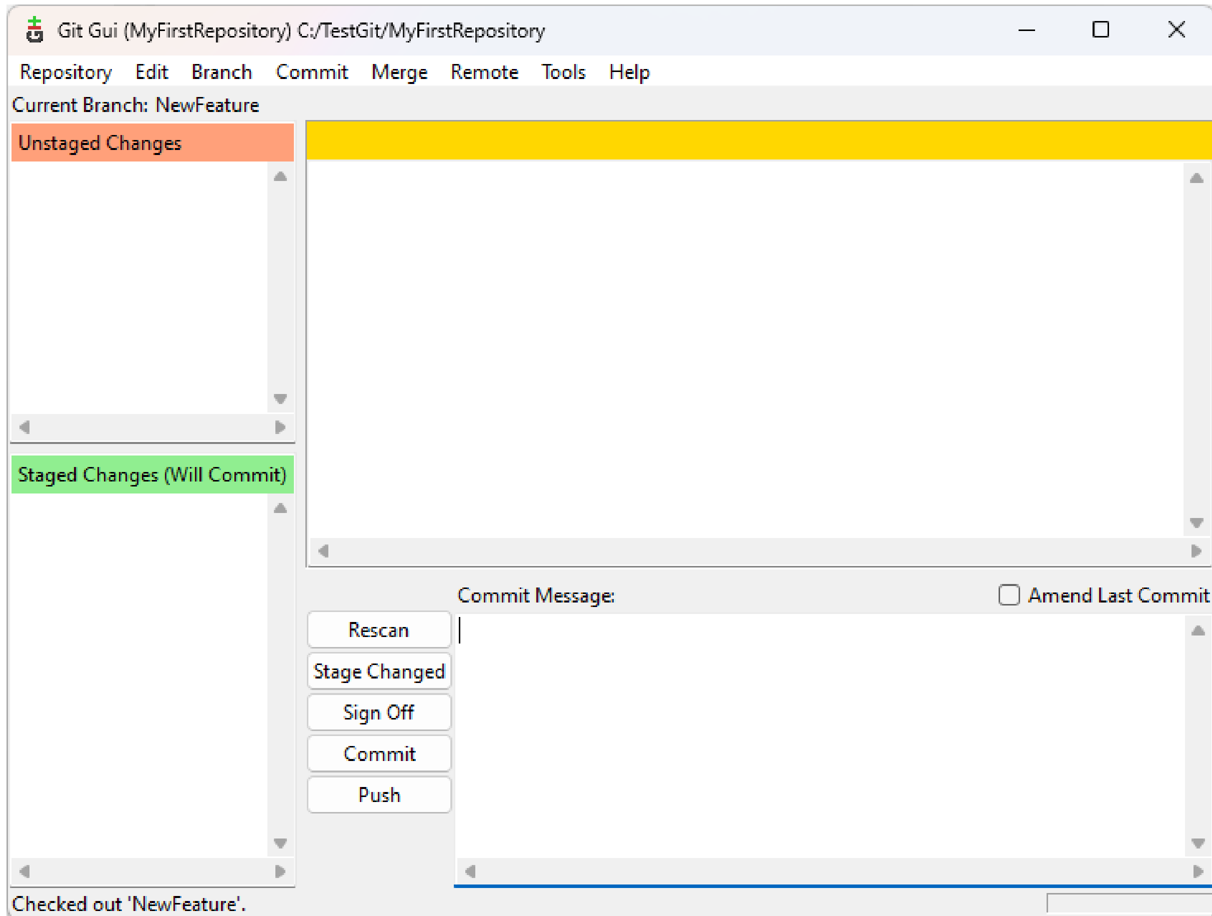


Figure 67: Create a branch from Git GUI.

You can see that it is already checked out to this new branch.

Let's change again our file:

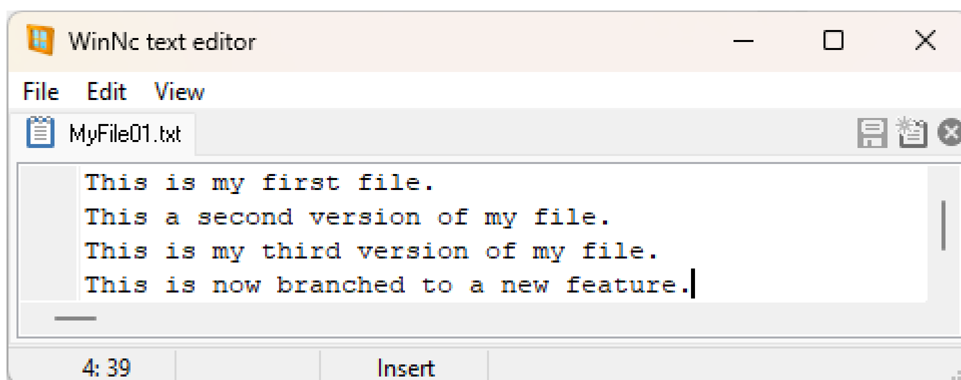


Figure 68: Change a file in the new branch.

Let hit rescan to refresh the Git GUI and let's stage and commit the change.

Let's push the change:

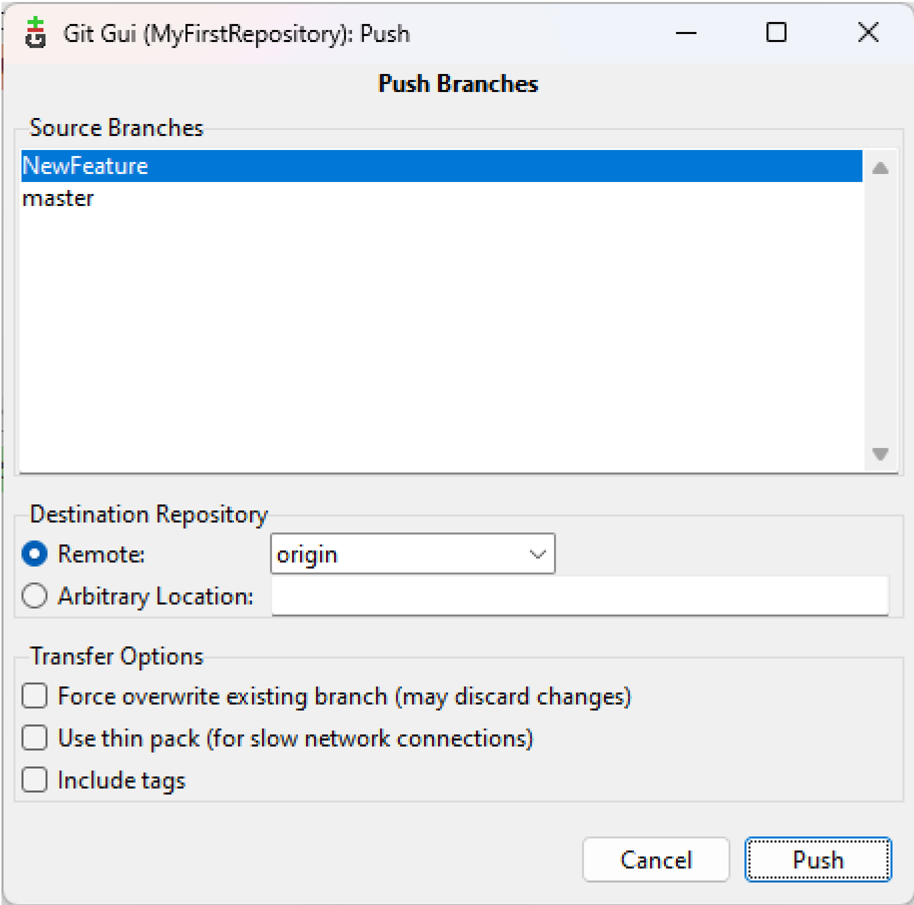


Figure 69: Push the branch.

Go to the Gogs GUI and refresh the repository:

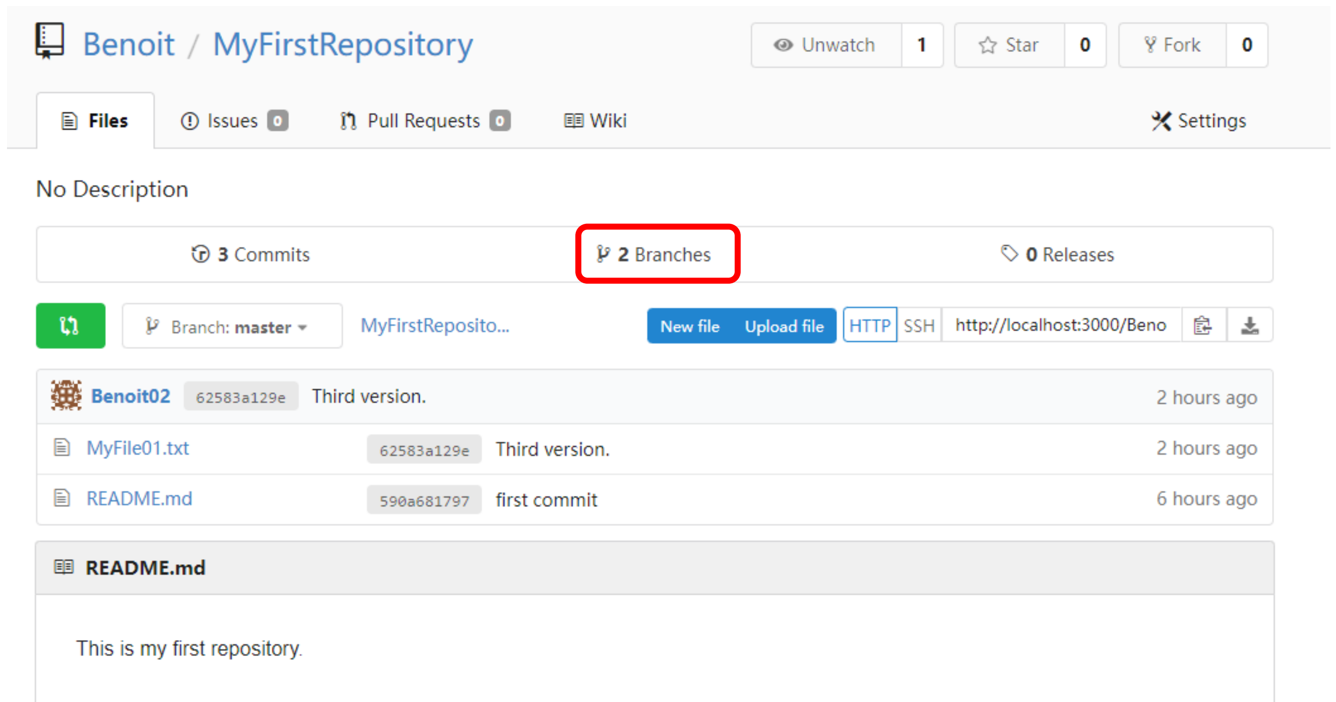


Figure 70: You see now two branches.

Change the branch to NewFeature:

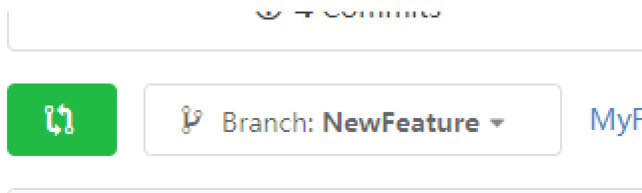


Figure 71: The NewFeature branch in Gogs.

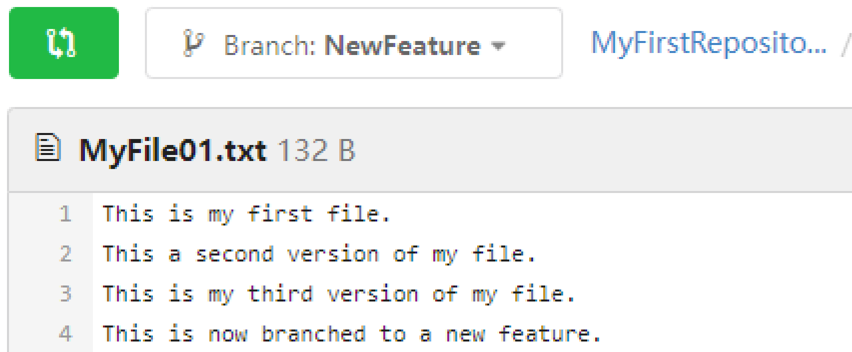


Figure 72: Changed file in the new branch.

Back to the main repository files, hit the green button:

Figure 73: The compare button.

By choosing the Compare field to NewFeature, you can now see this:

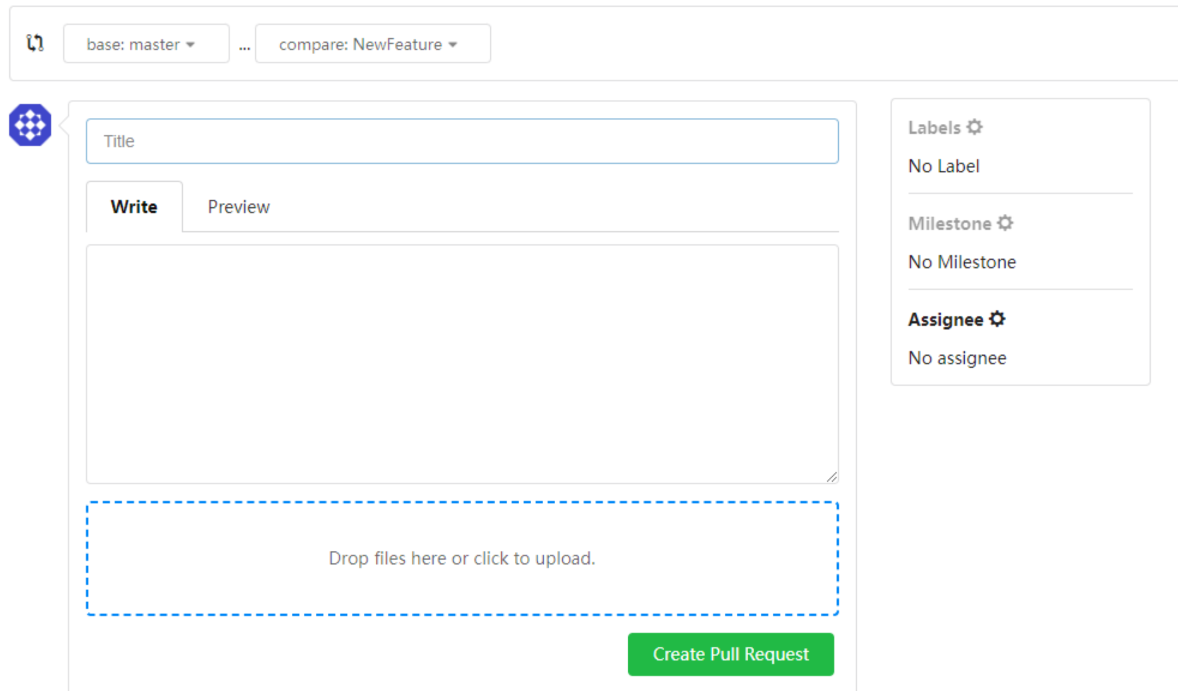


Figure 74: Compare branches, create Pull Request.

Fill and create the pull request:

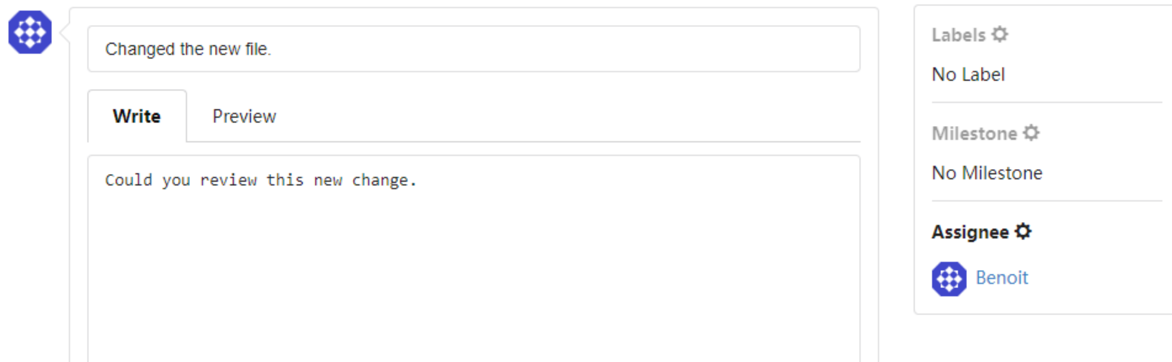


Figure 75: Pull request creation.

With this configuration you can only assign Benoit (yourself) to the pull request, since the repository has been created owned by Benoit, through no organization nor team.

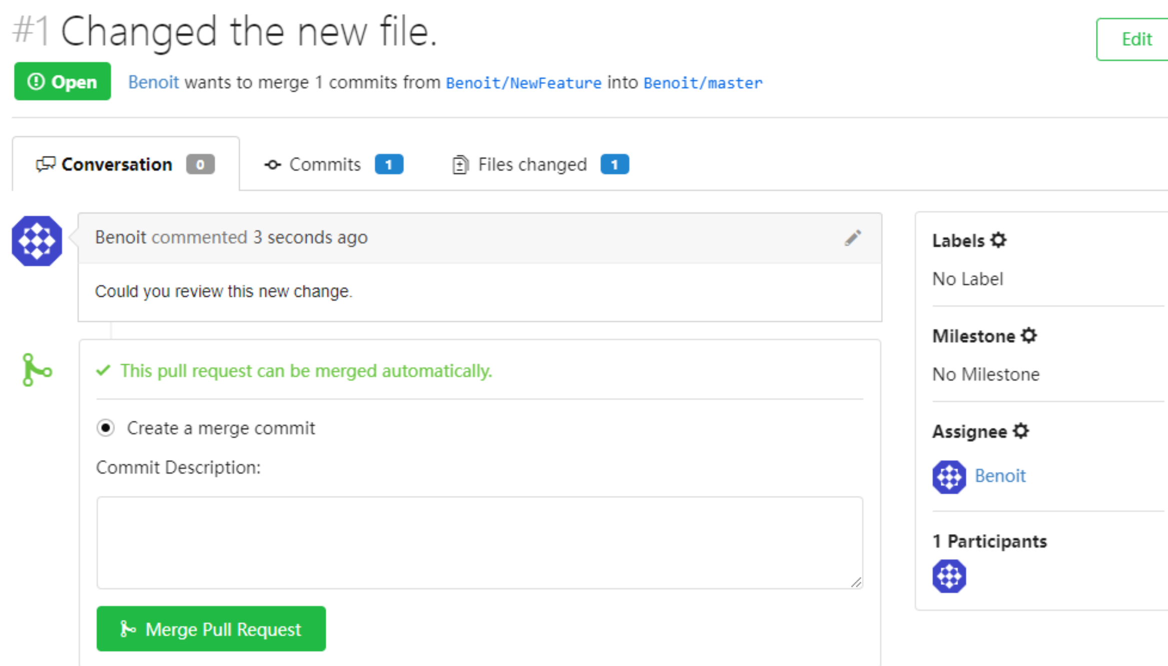


Figure 76: Pull request created.

You could go on discussing on this pull request.

Go back to the main repository screen:

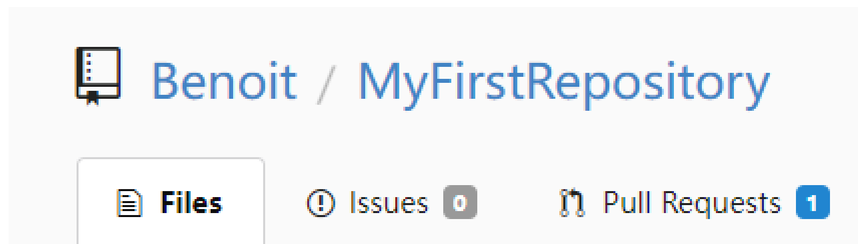


Figure 77: Pull request pending.

Go to this request and open it:

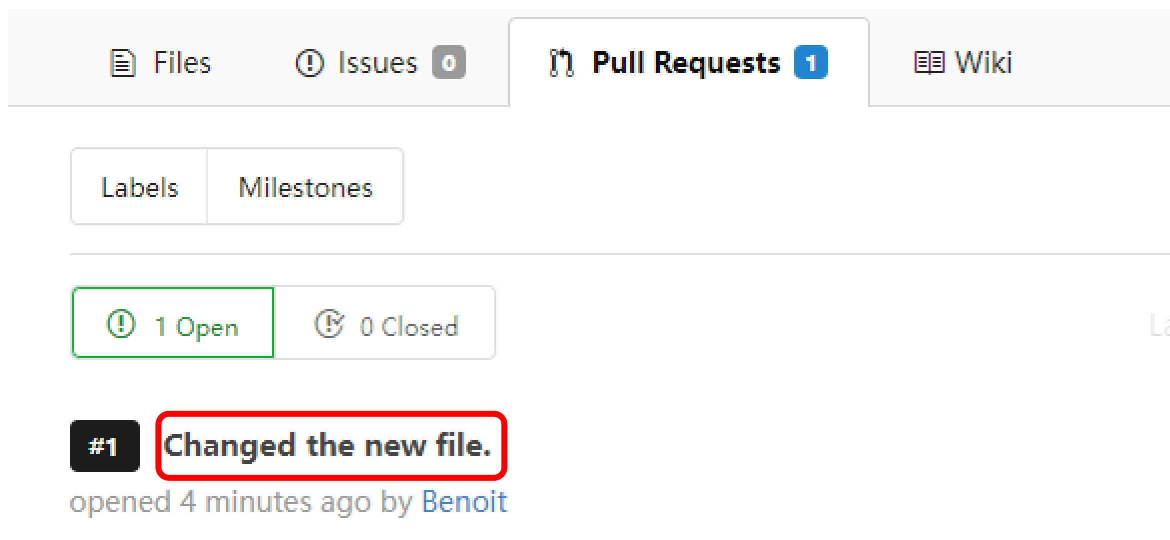


Figure 78: back to the pull request.

Merge it (hit the green “Merge Pull Request” button.

Back to the main repository branch, hit the MyFile01.txt file:

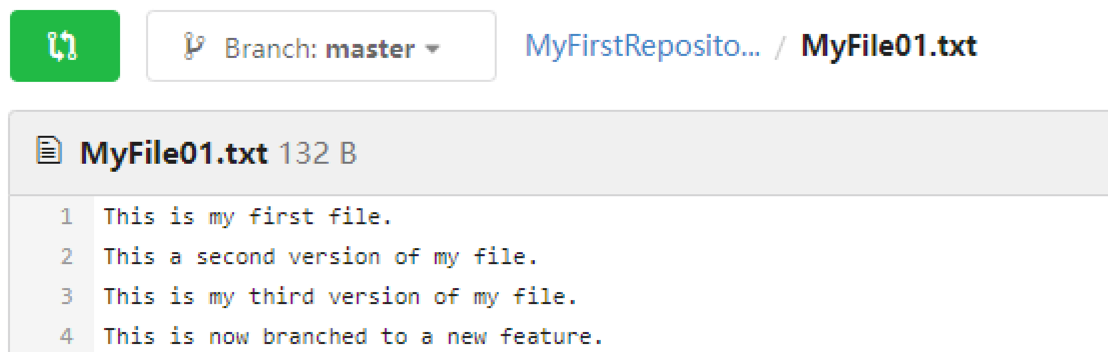


Figure 79: branched file in the master.

You see changed file has been pushed to the master branch.

Go back to the Git GUI, and check out to the master branch (Branch->Checkout):

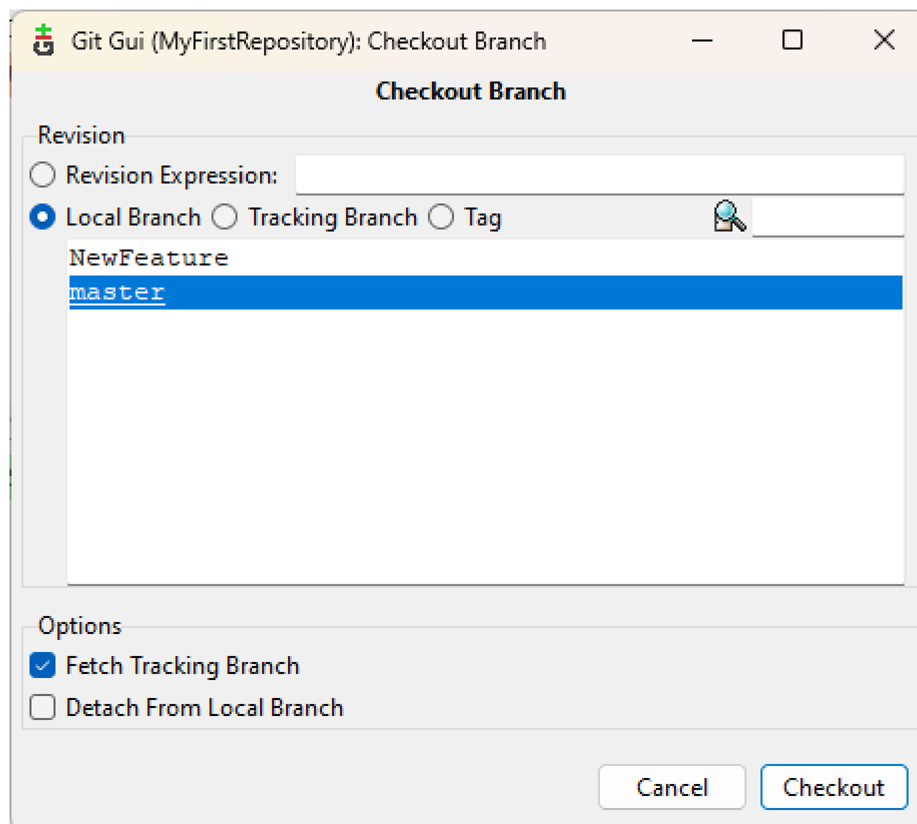


Figure 80: check out back to the master.

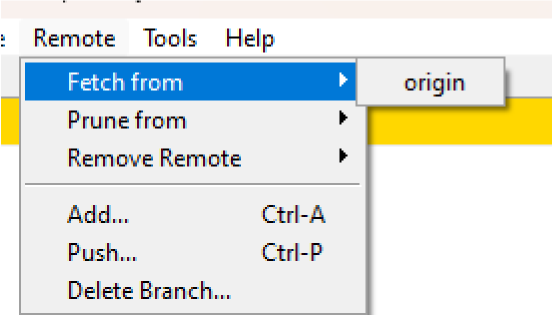


Figure 81: update the local master branch.

To refresh the master local branch, go to Remote->Fetch from origin.

If you view your local file, you see no change.

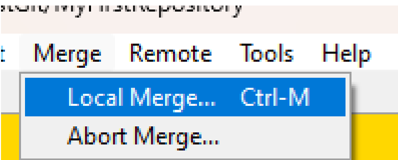


Figure 82: local merge.

But if you go to Merge-> Local Merge, then the file is correctly updated.

Connect the Gogs repository to an existing local one.

Create now a “c:\TestGit\MyFirstRepositoryCopy” folder.

From there, launch Git Bash.

Execute a git Init command.

Execute the first git remote command, but instead of a git push, execute a git fetch --all[44]:

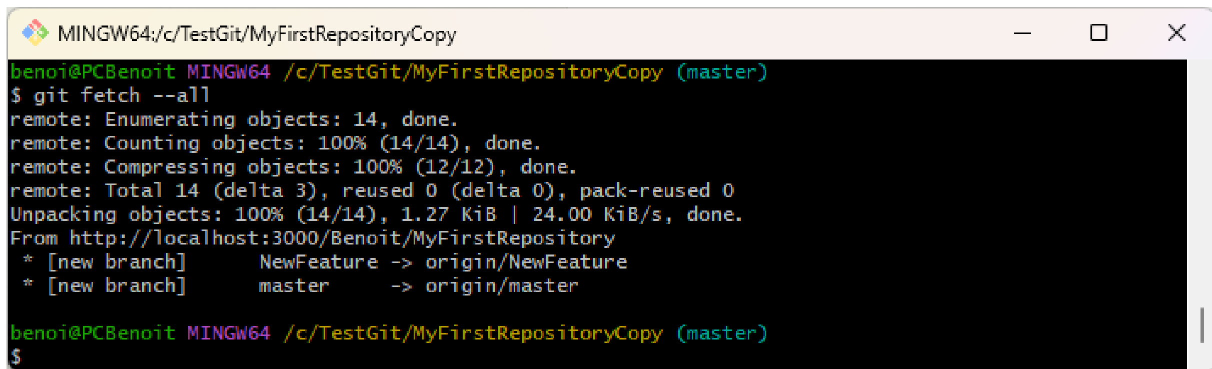


Figure 83: Git Fetch all.

Launch Git Gui in this new folder.

[44] Which actually gets all branches from the origin.

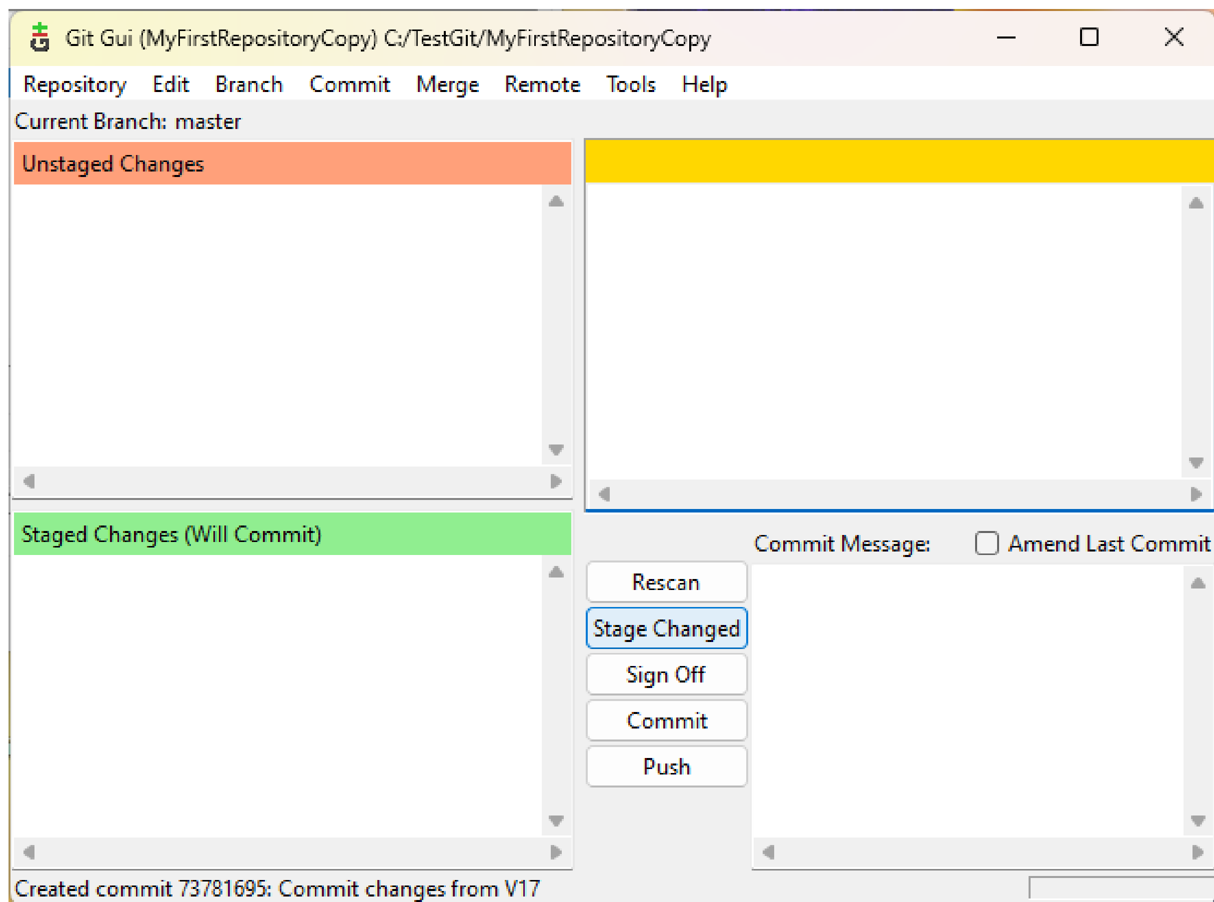
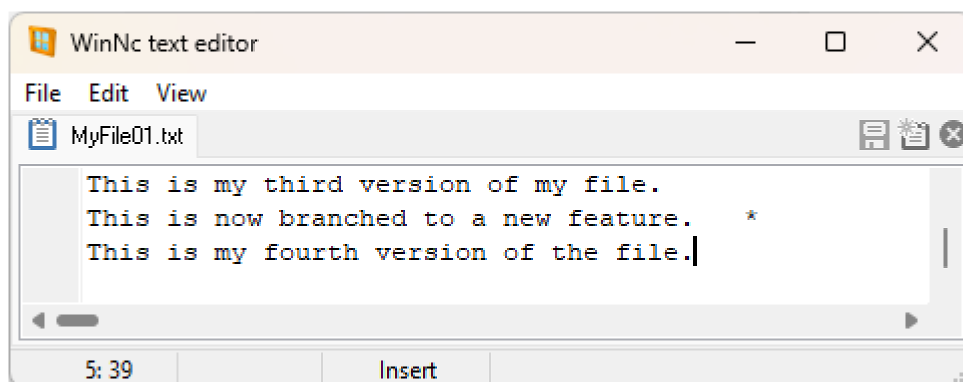


Figure 84: Launch Git Gui in a new folder.

Edit the MyFile01.txt:



And hit rescan to refresh Git Gui:

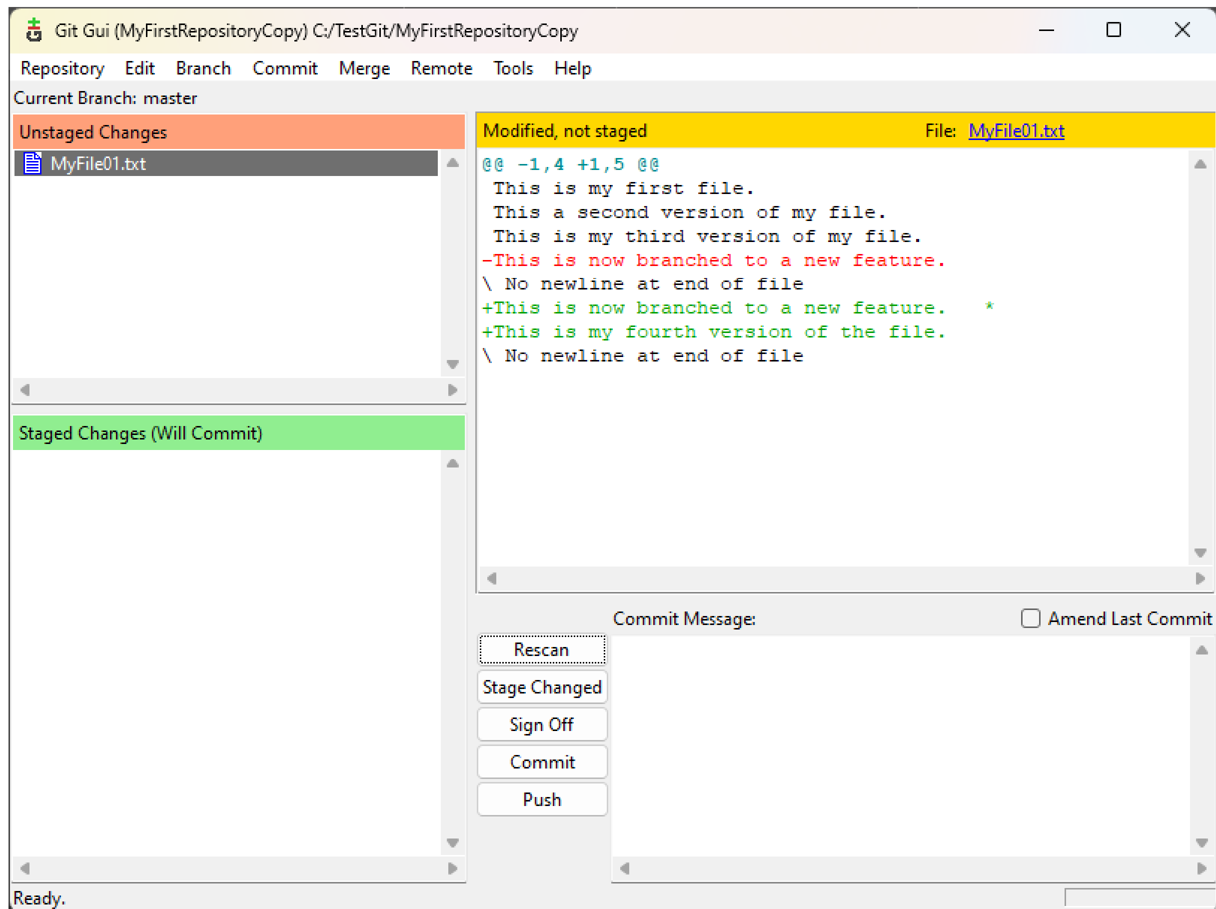


Figure 85: Change a file in a new copy.

Stage the change:

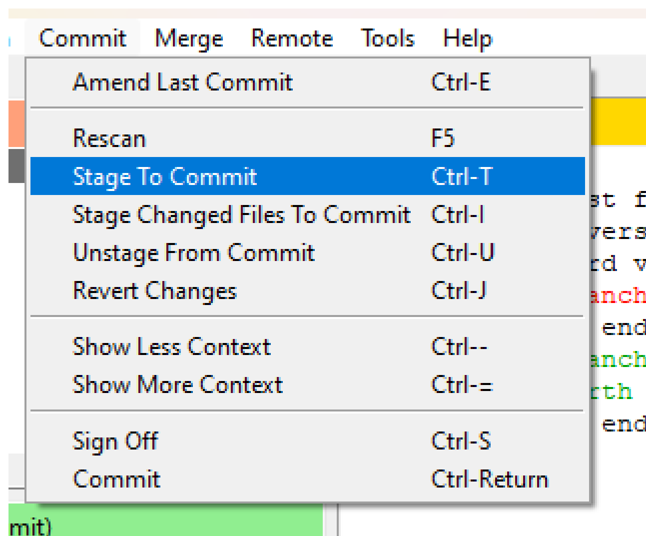


Figure 86: Staging a change.

Commit it:

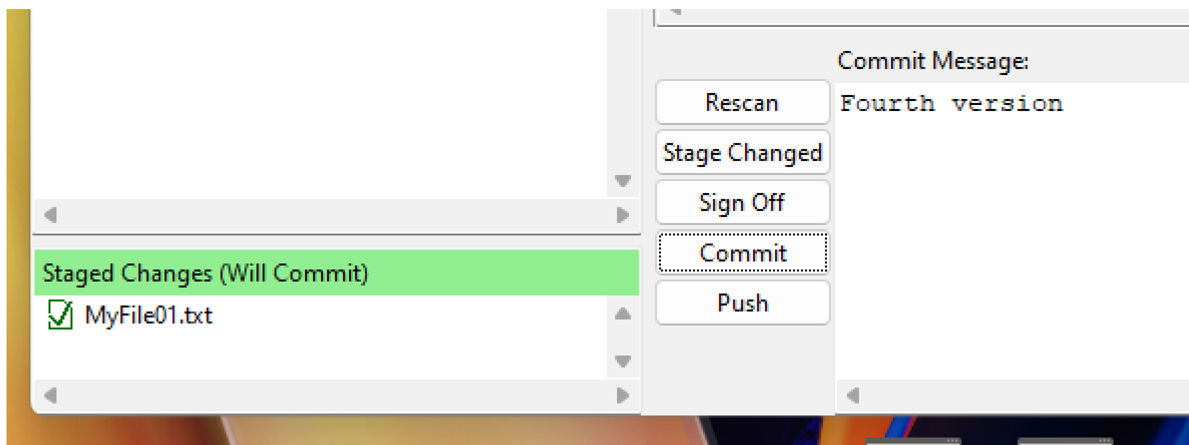


Figure 87: Commit the staged change.

Push it:

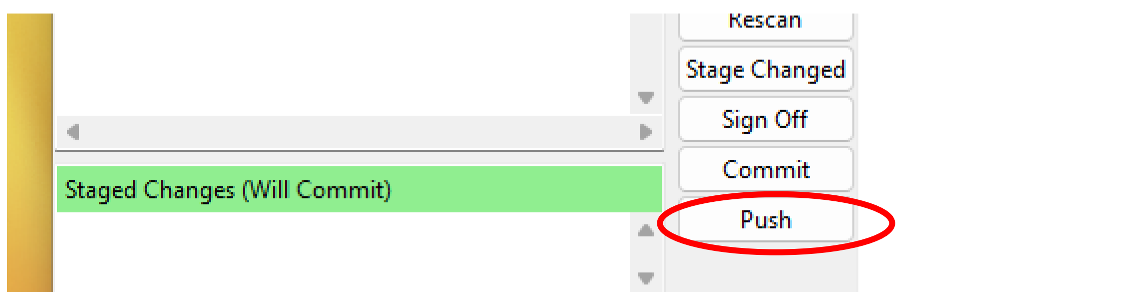


Figure 88: Push the change.

Acknowledge all popups.

Now you can see the change in the origin:

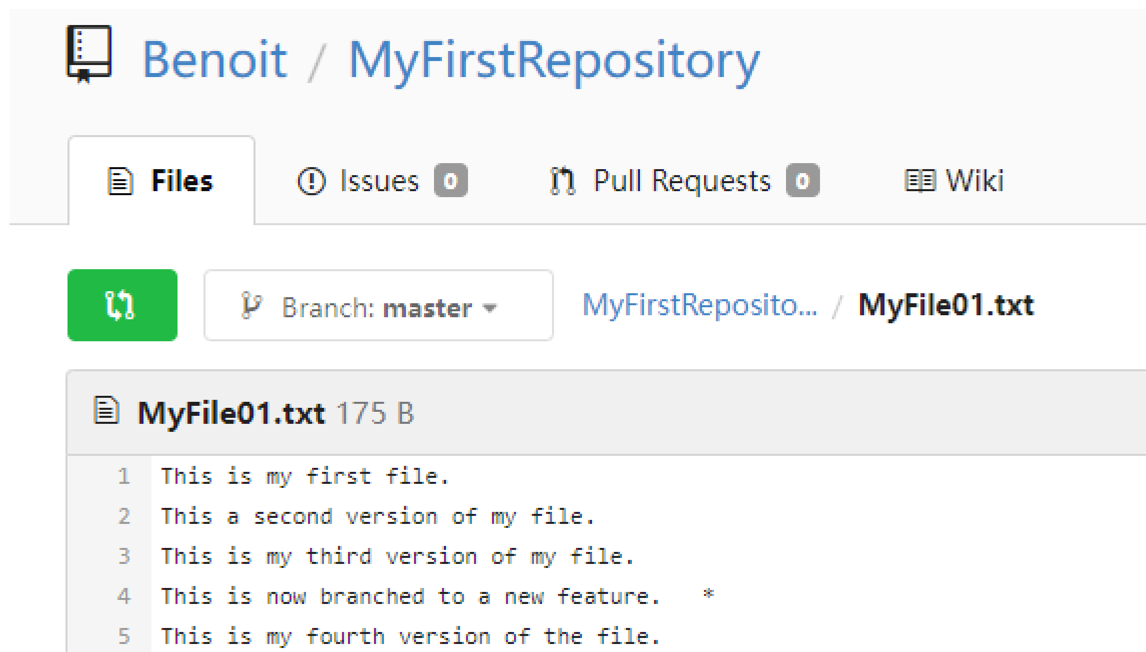


Figure 89: Change in the origin.

It pushed it to the origin's master branch. The NewFeature branch version stayed the same:

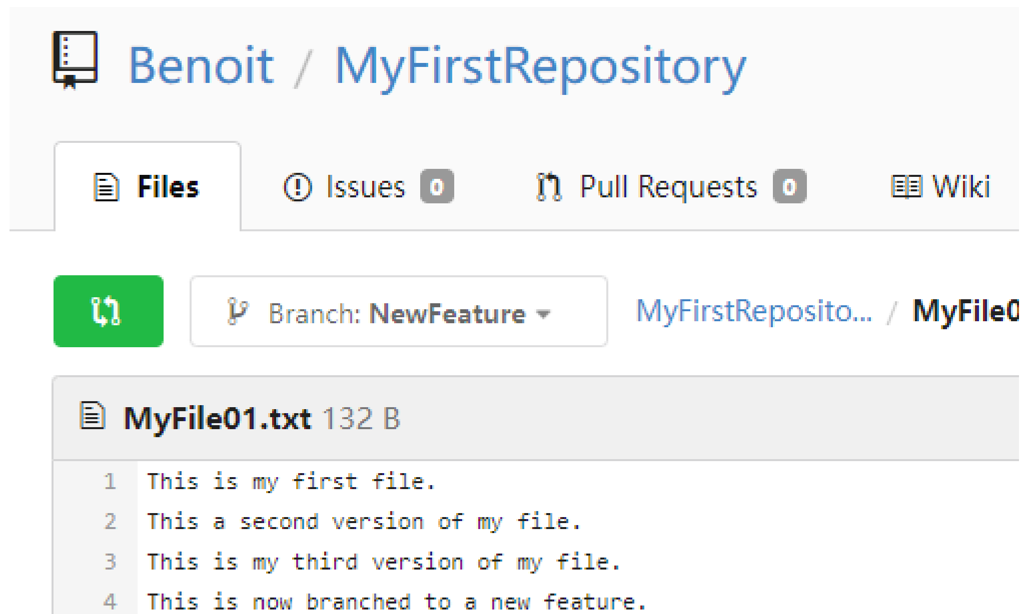


Figure 90: No change in the NewFeatureBranch.

Hit the compare button:

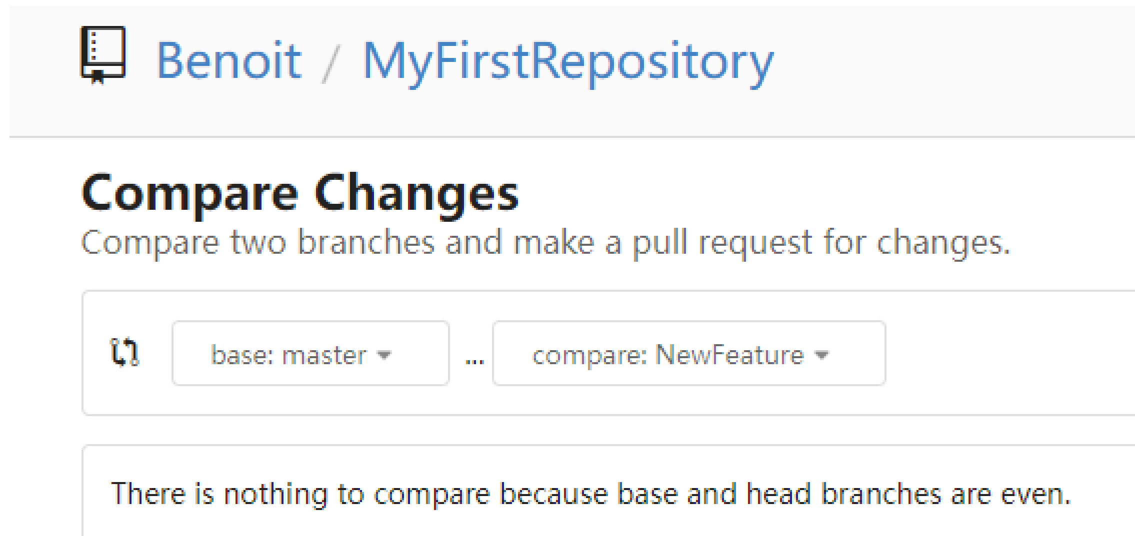


Figure 91: Compare.

Gogs detects no change.

Change the file again (fifth version), but before pushing from Git Gui, set the current branch to the NewFeature branch in Gogs.

Again, it pushes to the master branch.

This is all because when you launch Git Gui (or Git bash, or any other) in the current folder, it uses not only Windows environment variables but also the *index* of Git, which is together a branch and some location in the branch.

And currently the index is on master.

This can be done through Git Gui by using Branch->Checkout, which, behind the scenes uses the Git checkout command. Rather than doing this, will use the Git switch[45] command through Git bash:

```
MINGW64:/c/TestGit/MyFirstRepositoryCopy

benoi@PCBenoit MINGW64 /c/TestGit/MyFirstRepositoryCopy (master)
$ git switch NewFeature
Switched to a new branch 'NewFeature'
branch 'NewFeature' set up to track 'origin/NewFeature'.
```

Figure 92: Switch to a branch.

Since we are further on in master compared to NewFeature, we merge master into the NewFeature branch:

---

[45] Git switch has been introduced in 2019, because Git checkout, on top of changing the index can also fetch files form the starting branch, which might be confusing. Git switch does not alter nor add any file in the branch we want to connect to.

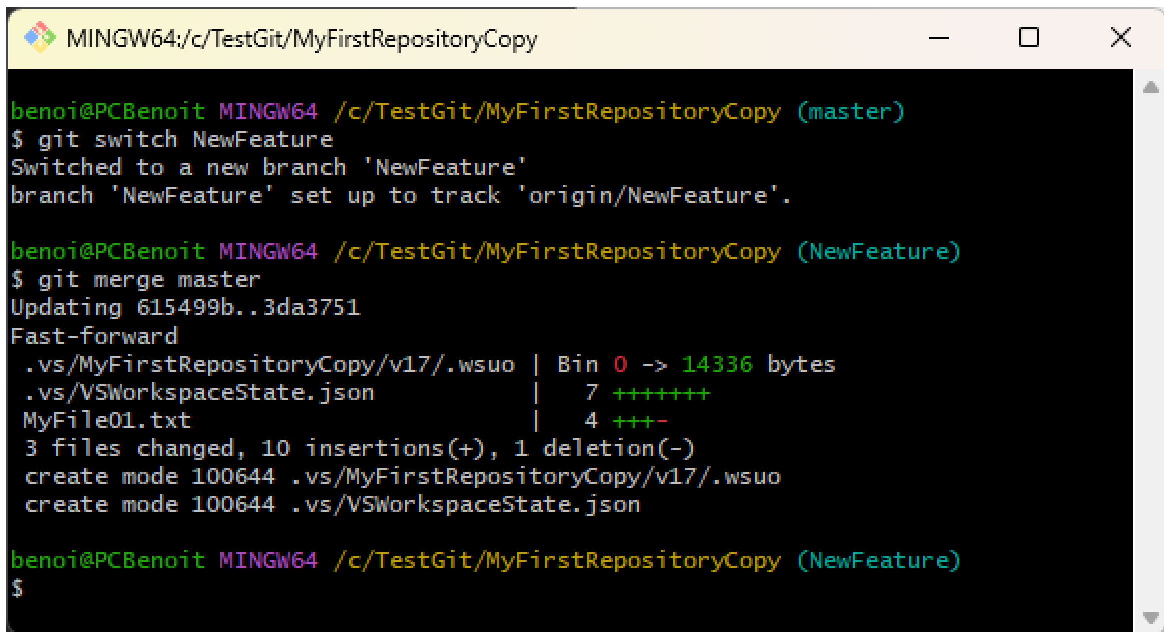


Figure 93: Merge the master into the current branch.

We see now that MyFile01.txt has been updated:

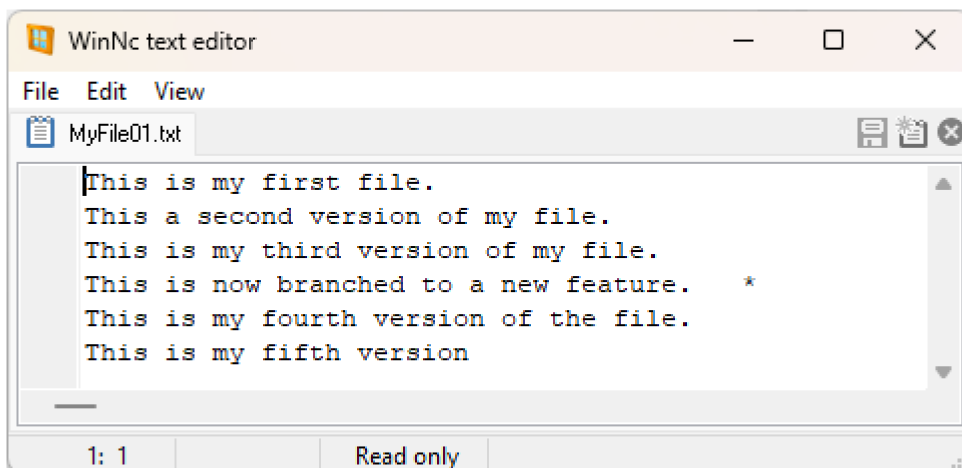


Figure 94: NewFeature has been updated.

Let's update MyFile01.txt to the sixth version (change the file content), stage and commit it:

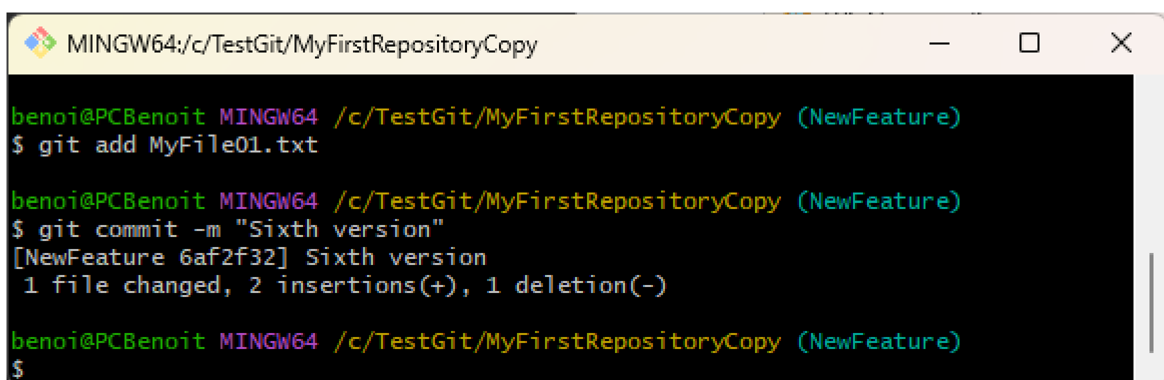


Figure 95: Update a file in the current branch.

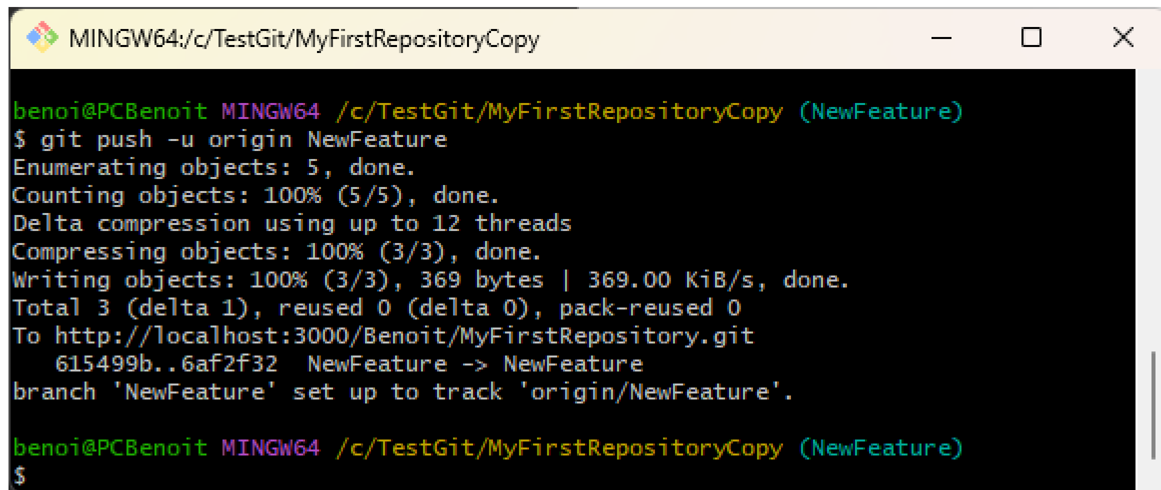


Figure 96: Push the branch to the remote.

You can see that the file has been updated to the correct branch:

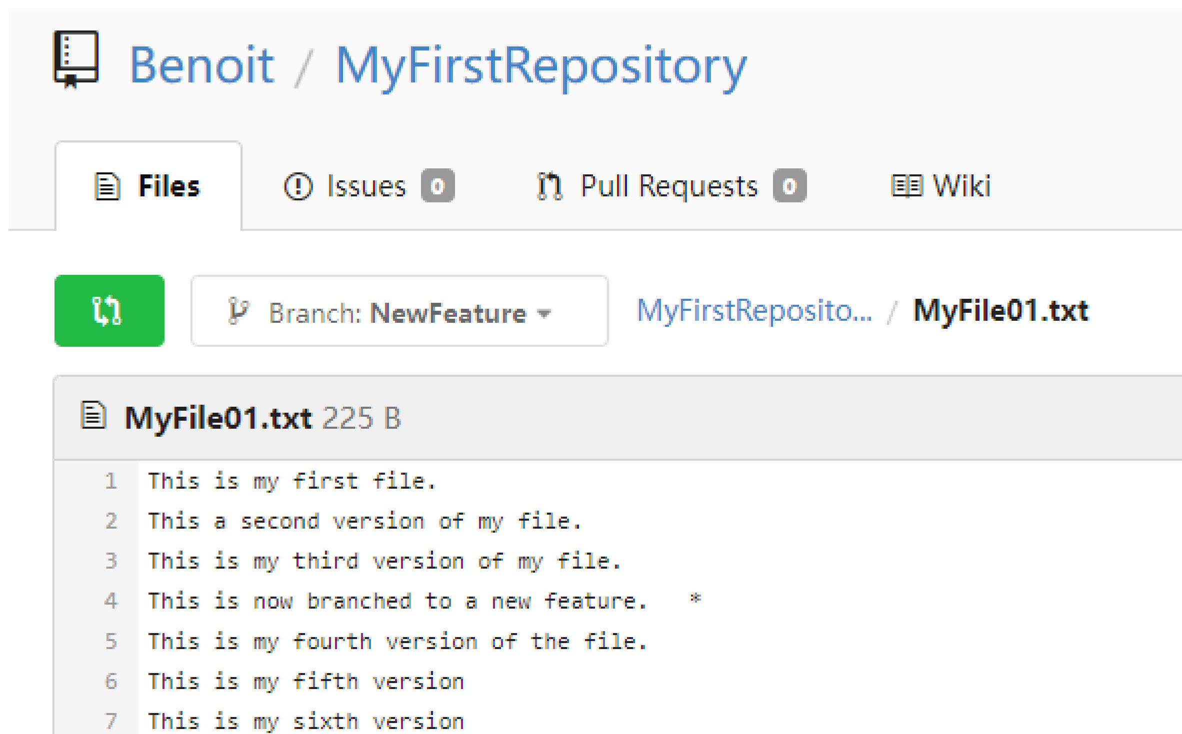


Figure 97: File updated in Gogs.

If we decide the new feature development to be finished, and the origin is up to date, we can delete the NewFeature branch to be obsolete, we can delete the remote branch[46]:

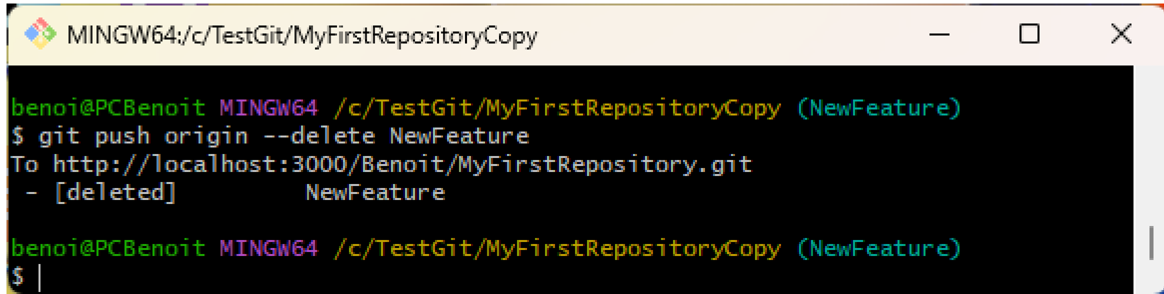


Figure 98: Delete the remote branch.

Which results in Gogs in having just one master remaining branch:

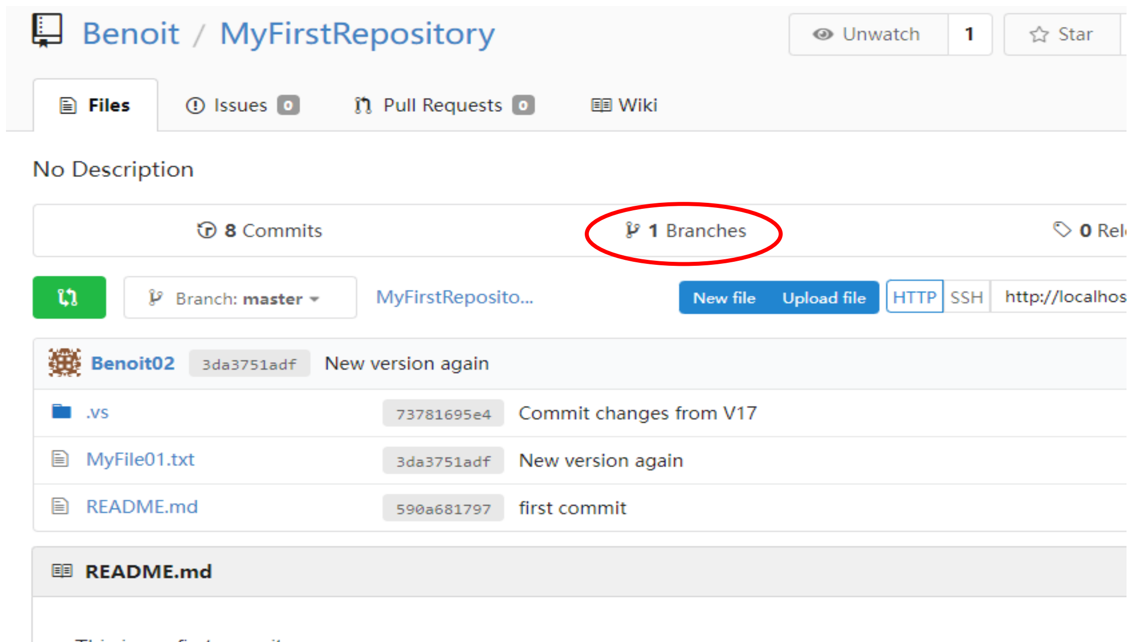


Figure 99: One remaining branch after push and delete.

[46] Since there is no way to do it directly in Gogs, this is the only way to proceed.

The fork process.

First let's create the organization dedicated to the collaborators:

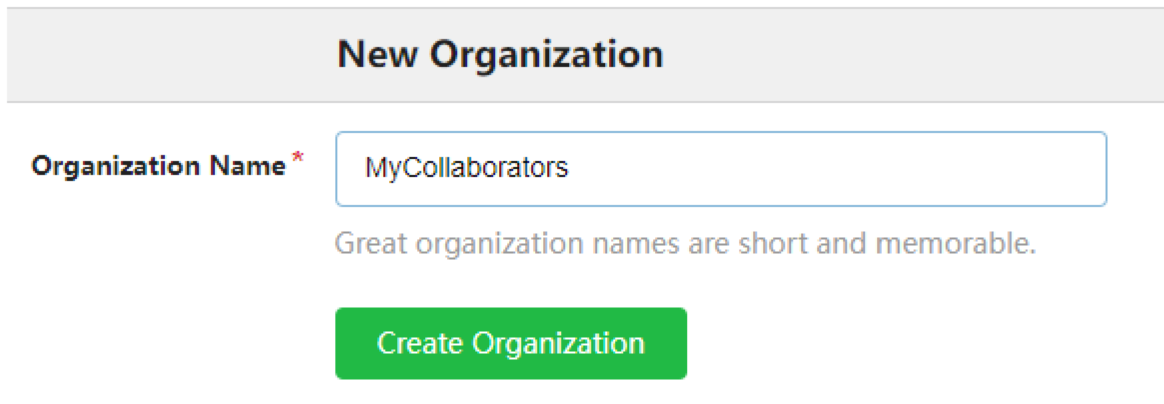


Figure 100: Collaborators organization.

Then let's connect to it:

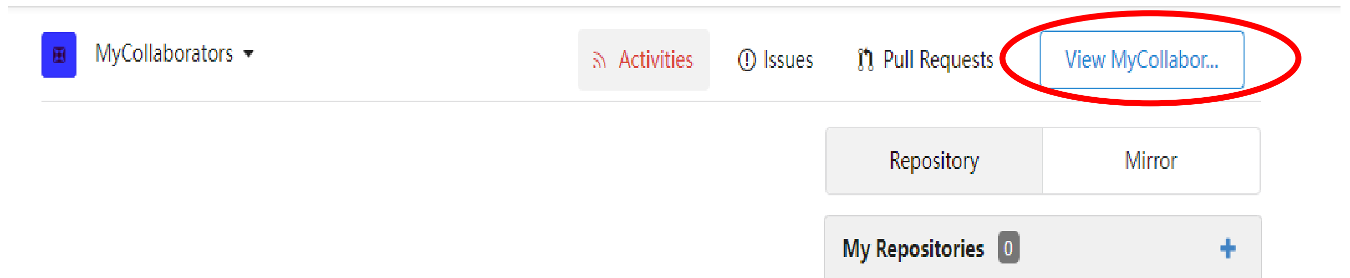


Figure 101: Connect to the collaborators organization.

Let's create a team associated to the organization (you could as well directly invite people):

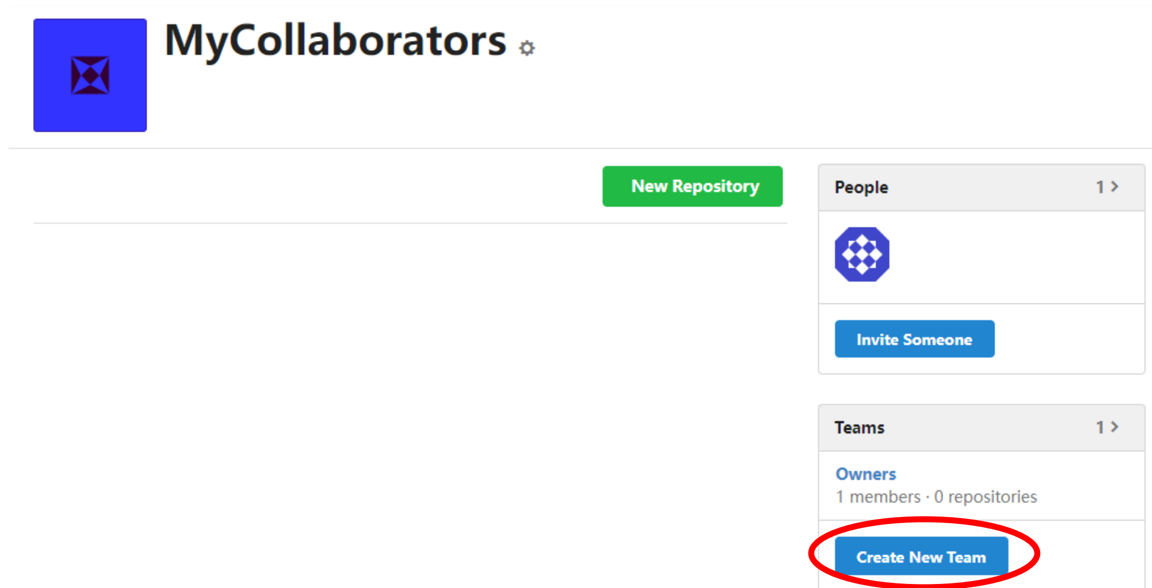


Figure 102: Team associated to the organization.

Give write access to repositories that we will create in this team:

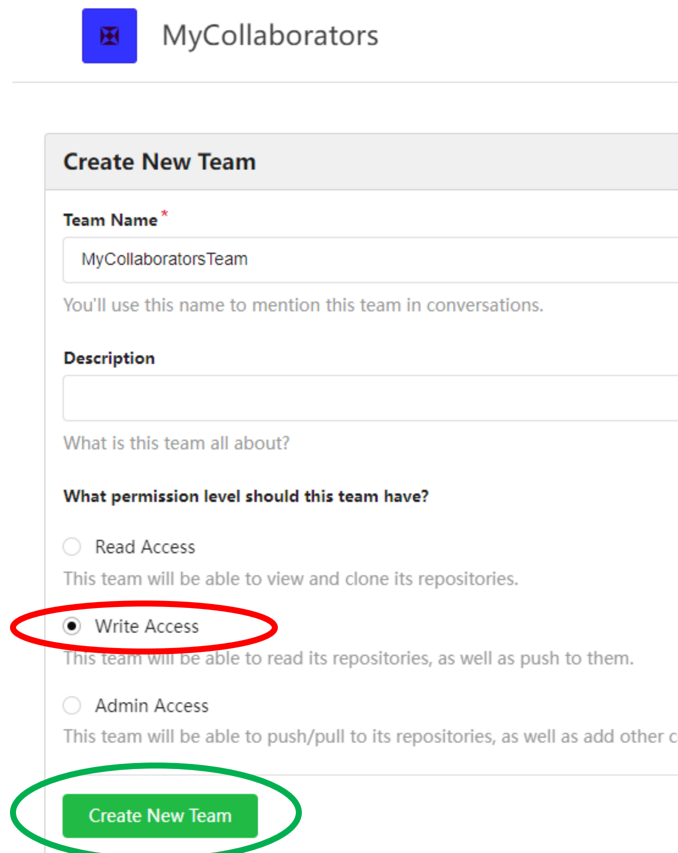


Figure 103: Collaborators team.

Add members to it:

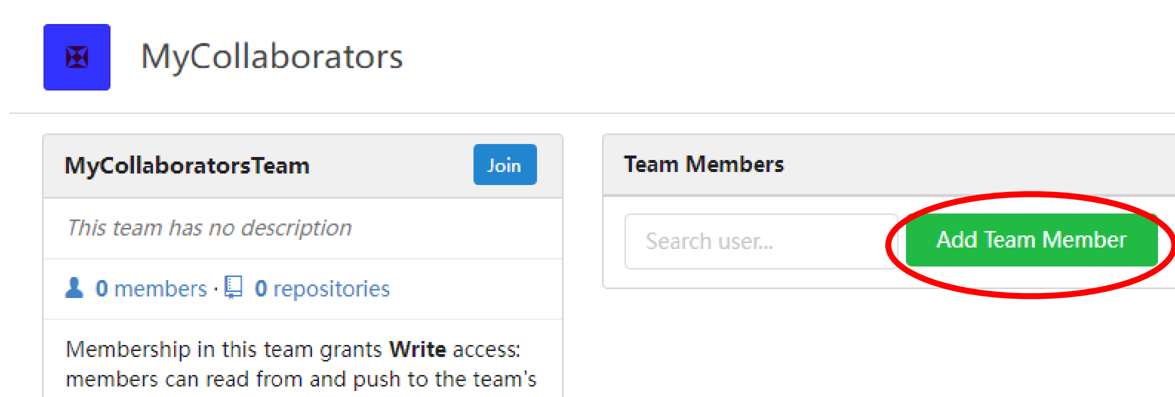


Figure 104: Add members to the collaborators team.

The full collaborators team:

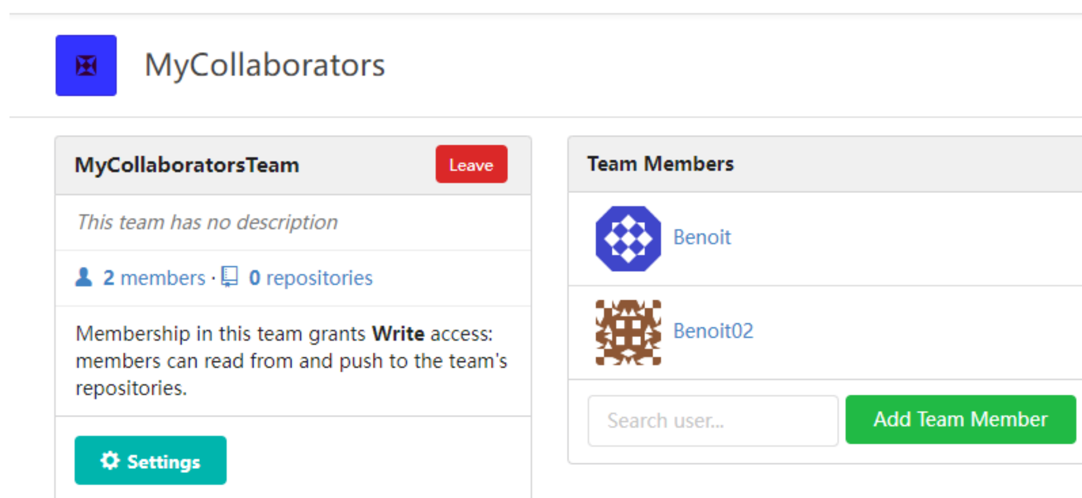


Figure 105: The collaborators team.

Now let's create the main repository:

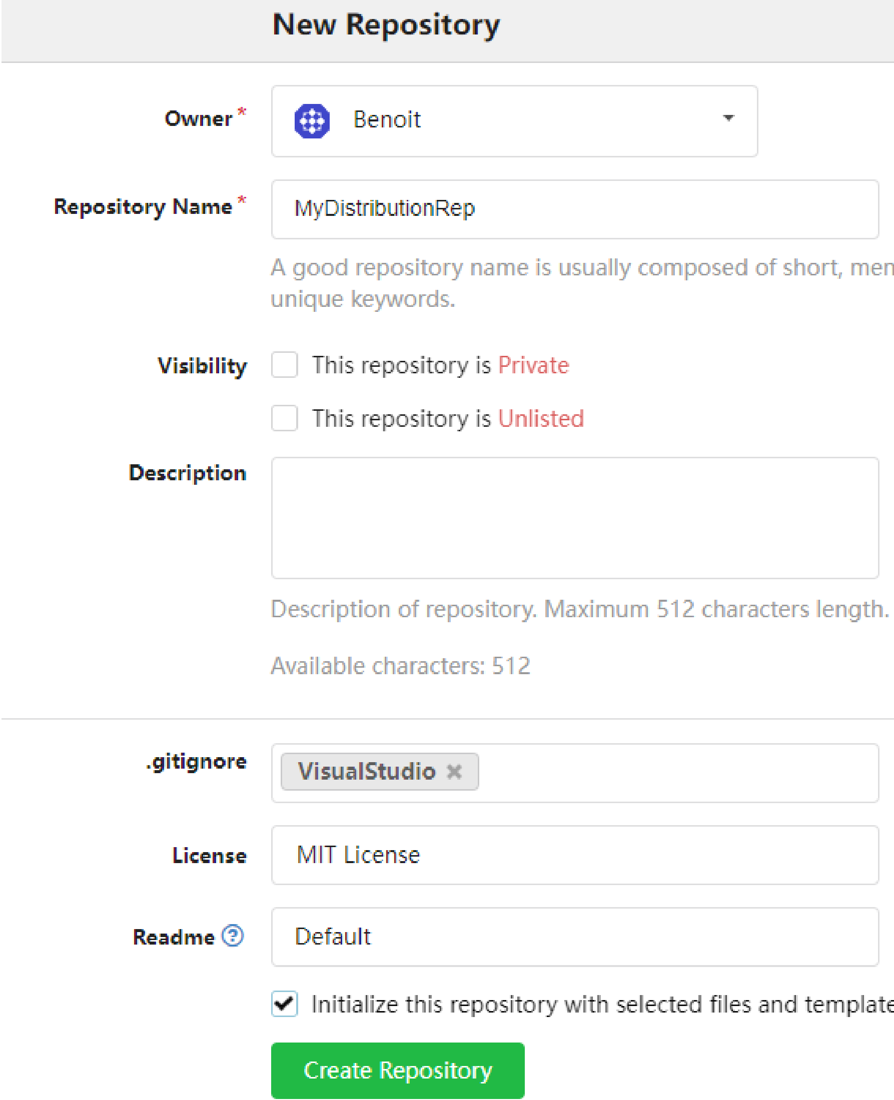


Figure 106: The main repository.

Let's populate the repository (through direct file import)[47]:

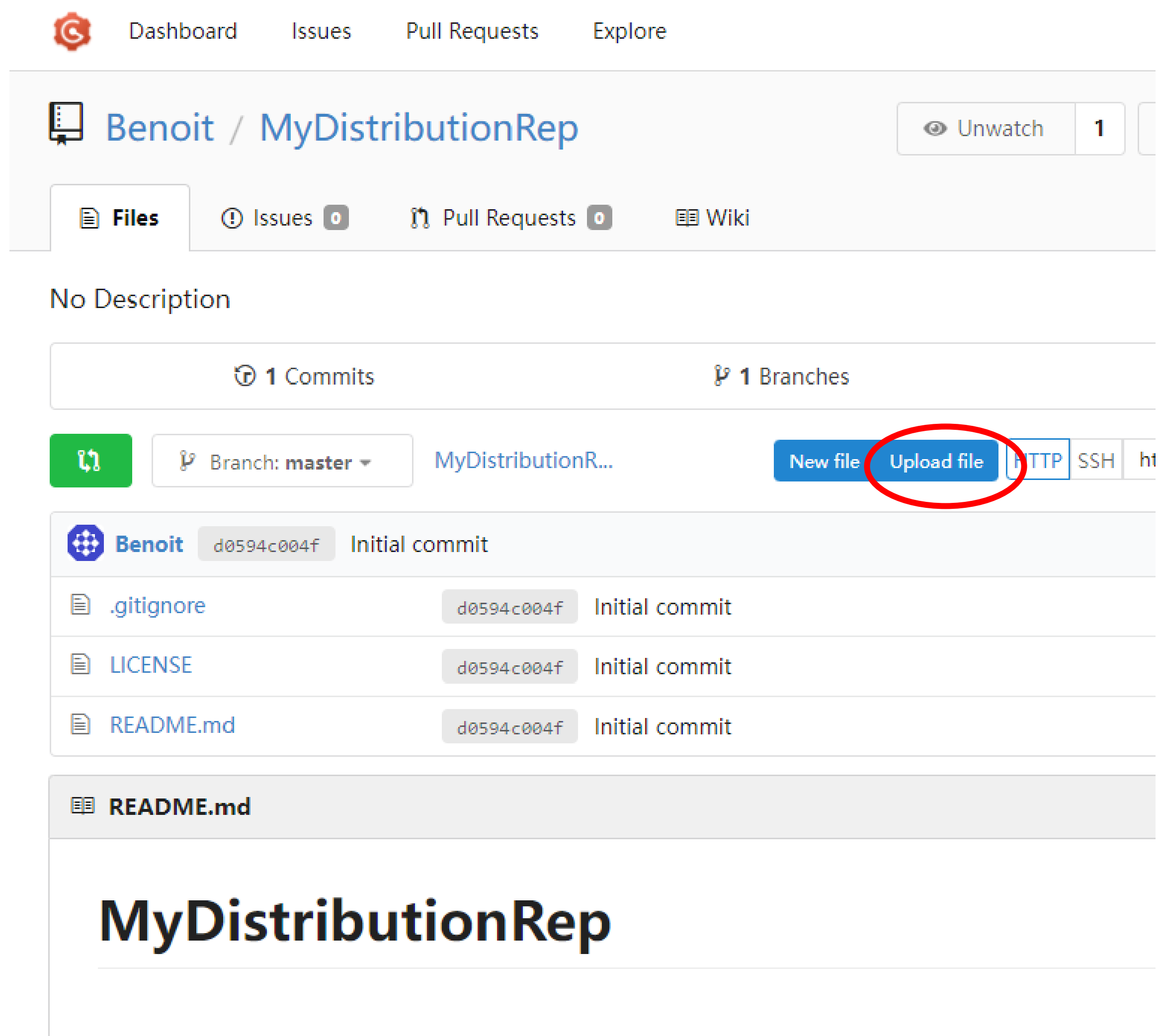


Figure 107: Repository population.

Let's drag and drop files (in our case MyFile01.txt) to the repository:

[47] You could as well populate it by connecting it to an external repository, as showed above in this chapter.

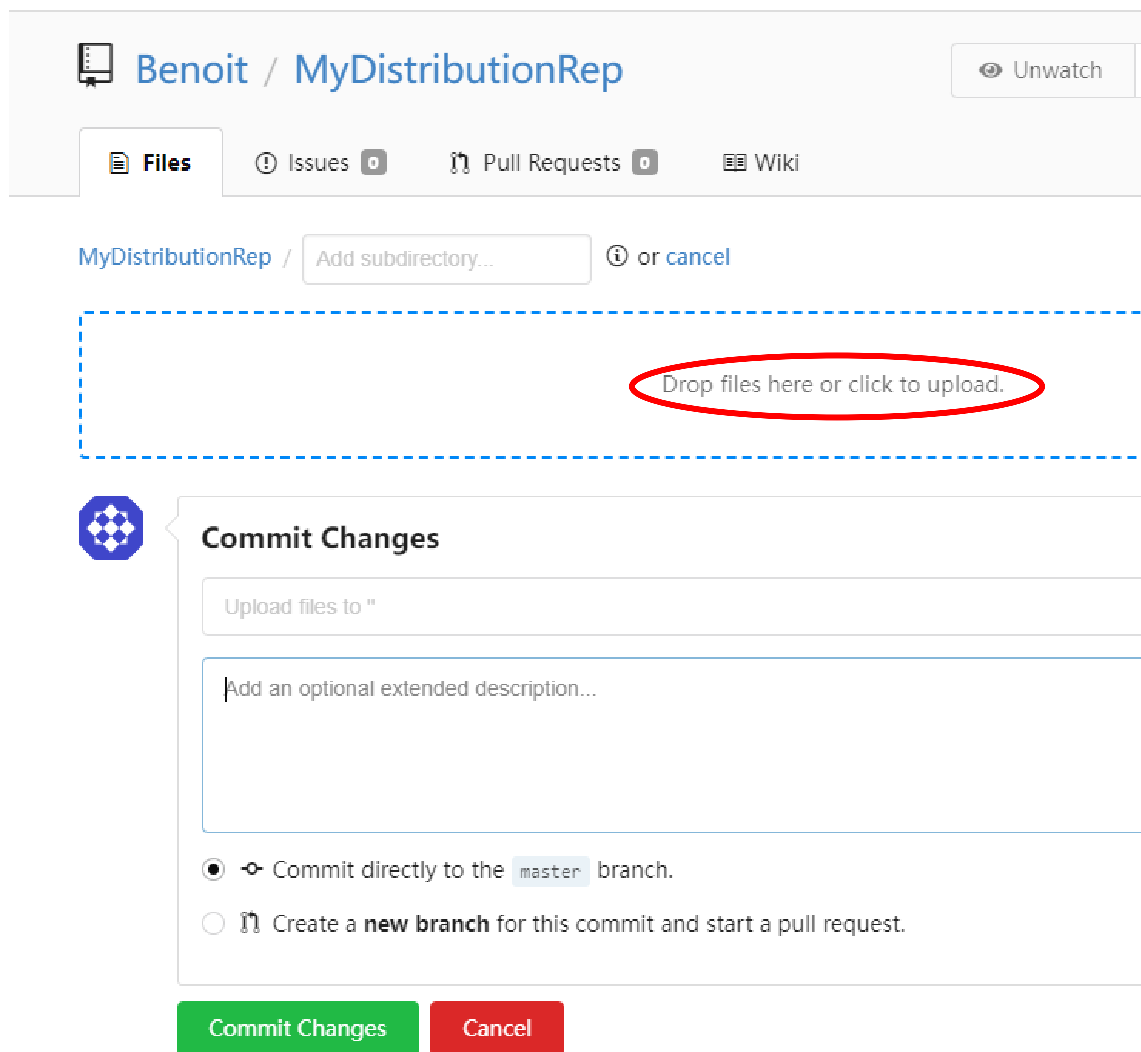


Figure 108: Drag & drop to the main repository.

And, after the drop, add a message and commit the addition:

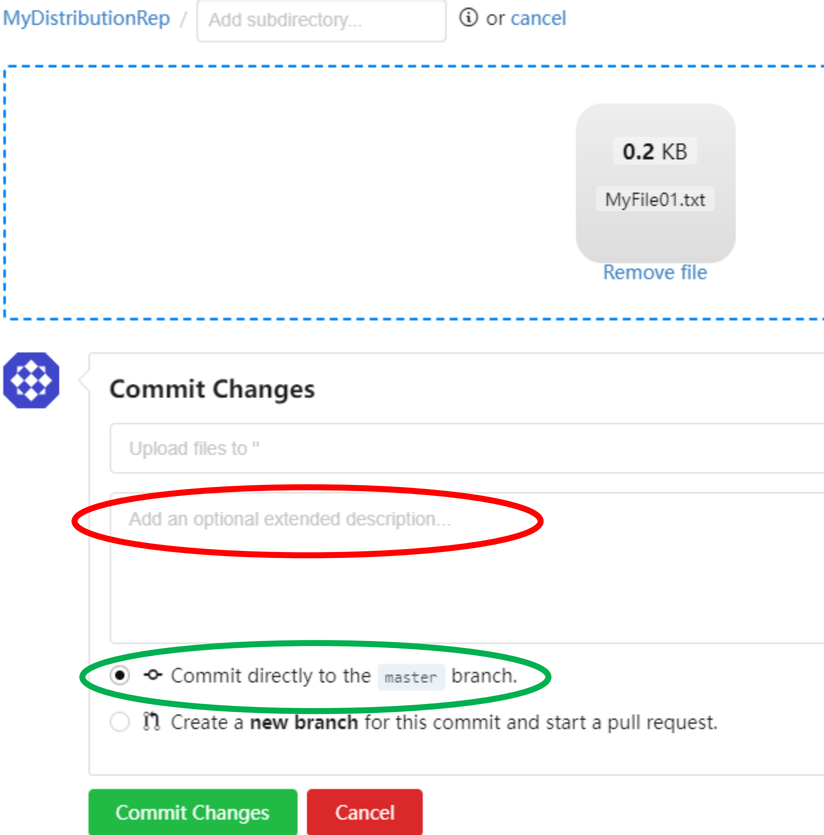


Figure 109: Commit the file addition.

The repository populated:

Figure 110: The repository populated.

Now from the fork button, fork the main repository:

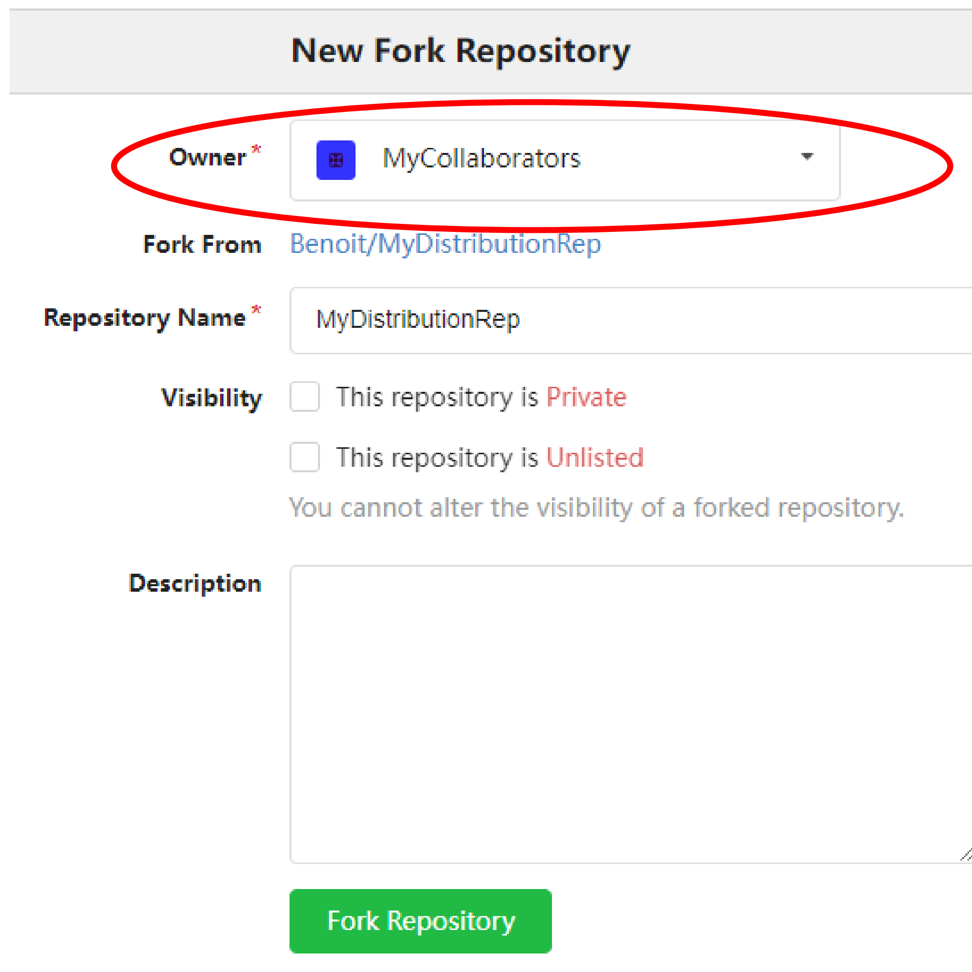


Figure 111: Fork the main repository.

Use the MyCollaborators organization as being the owner of the fork repository. Notice the two repositories:

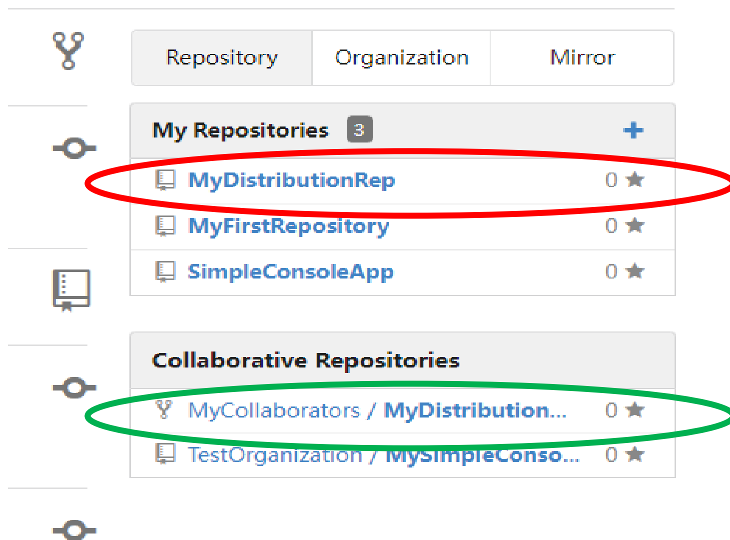


Figure 112: Main and collaborative repositories.

The collaborative repository:

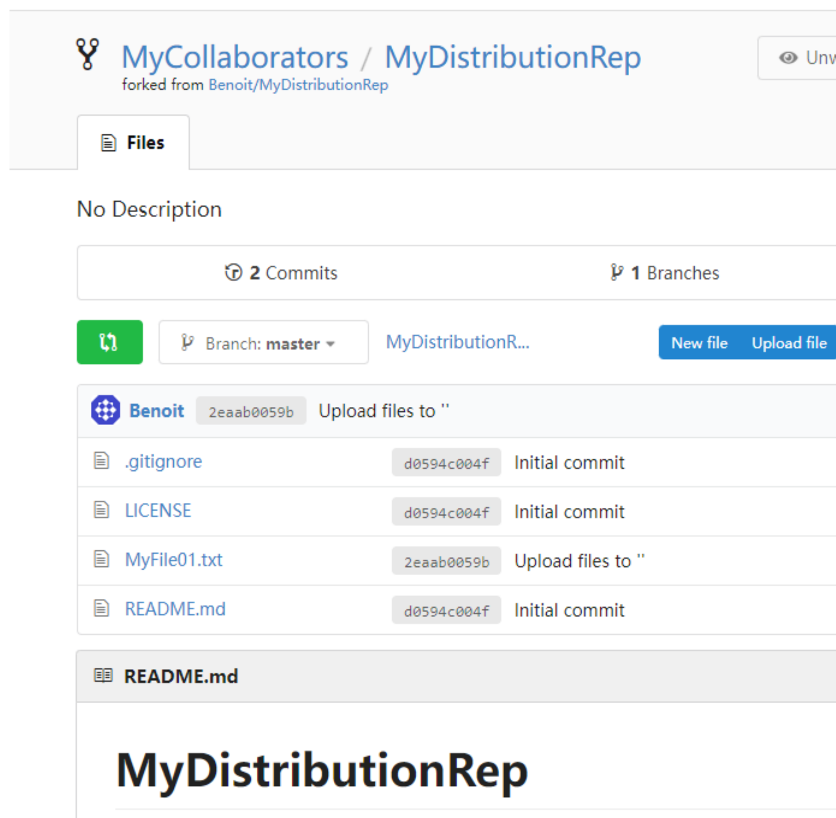


Figure 113: The collaborative repository.

Let's clone this repository:

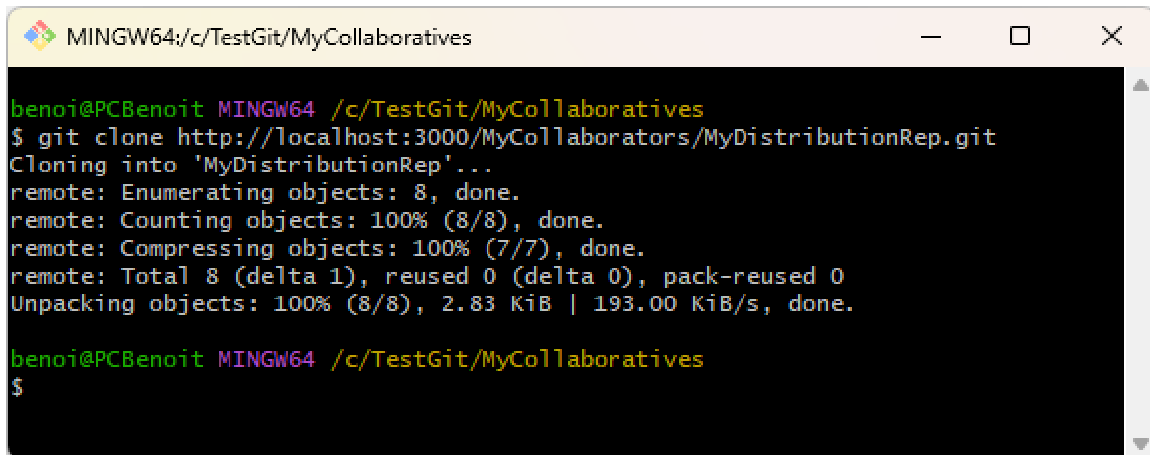


Figure 114: cloning the collaborative repository.

Let's create a NewFeature branch to it:

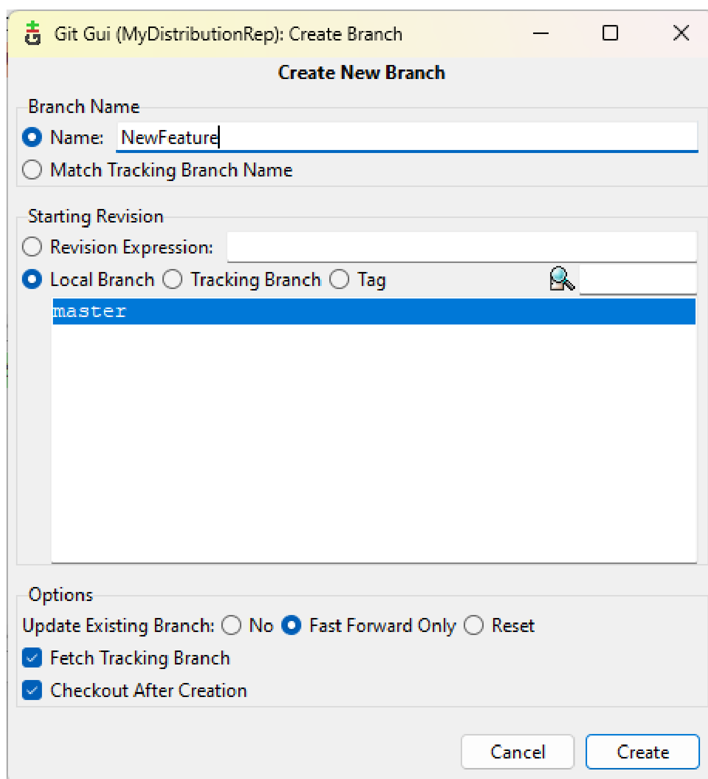


Figure 115: Create a NewFeature branch.

Let's modify MyFile01.txt [48]:

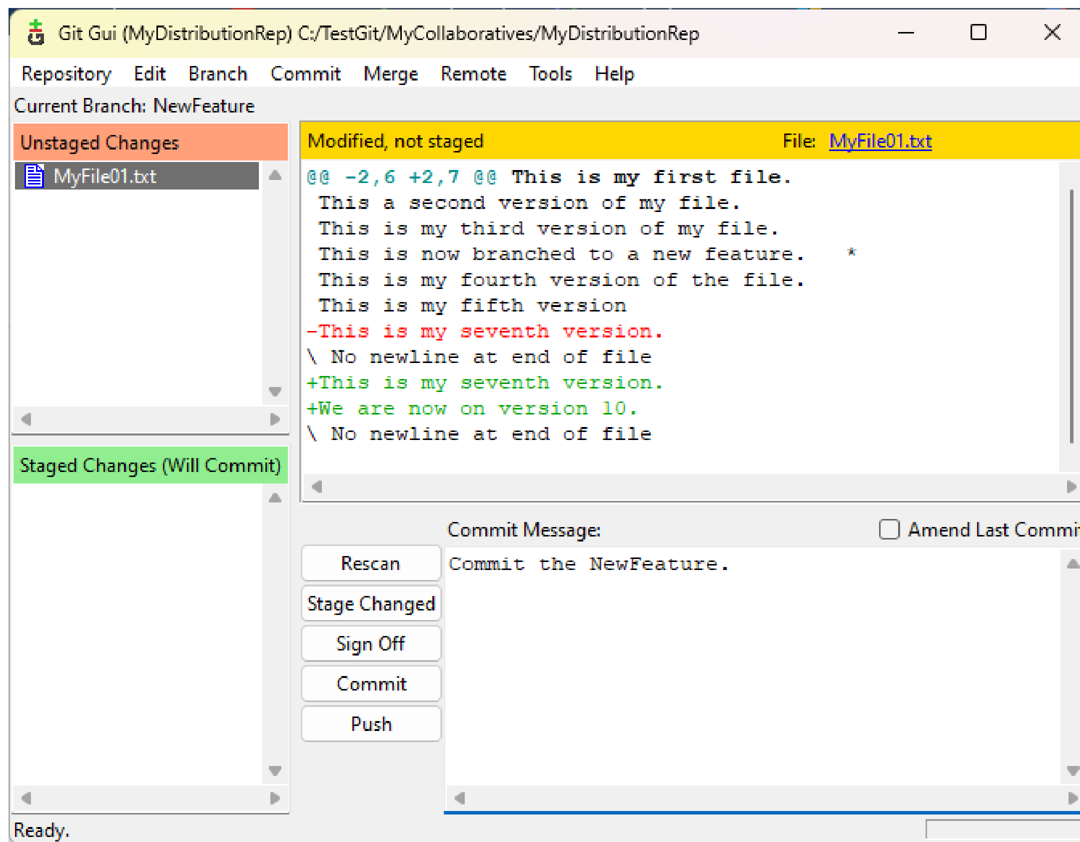


Figure 116: Modify MyFile01.txt on the NewFeature branch.

Stage the file, commit it.

Push it now to the origin:

[48] Remember : hit the Rescan button to see it.

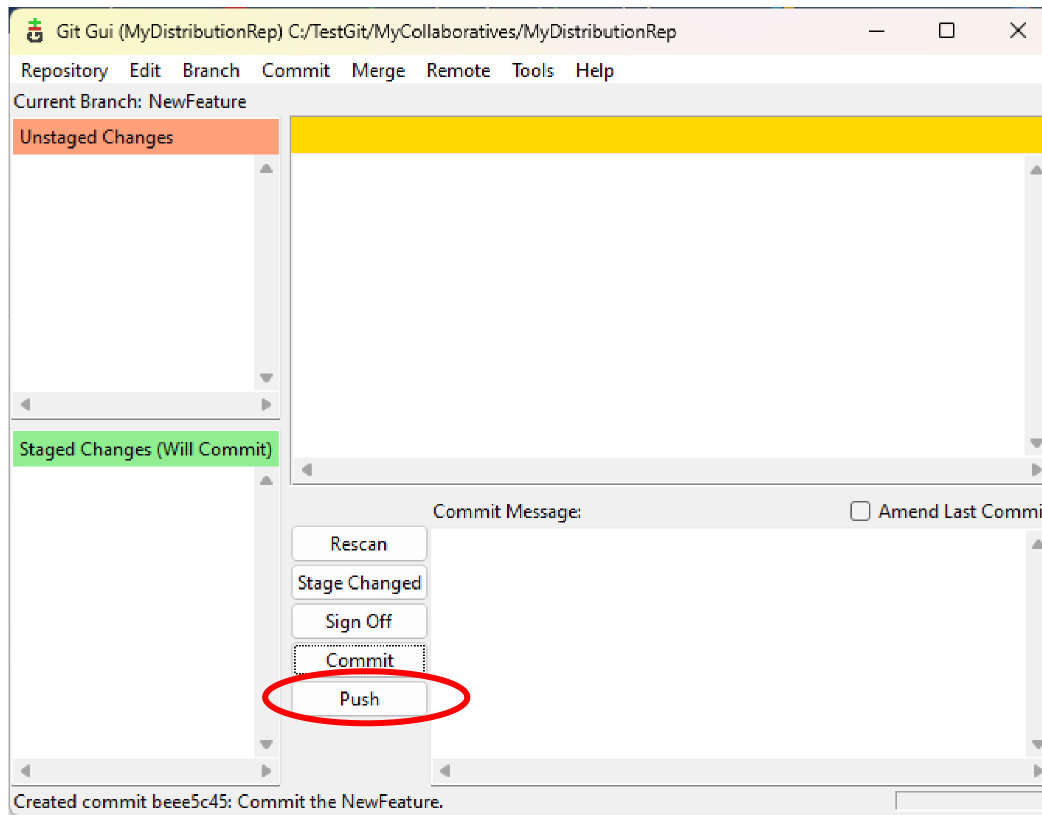


Figure 117: Push the NewFeature branch to the origin.

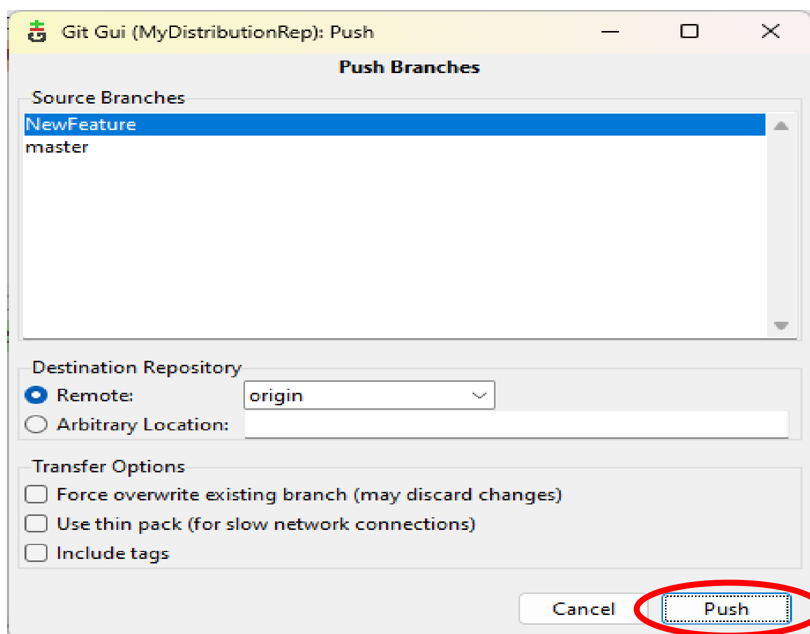


Figure 118: Push popup.

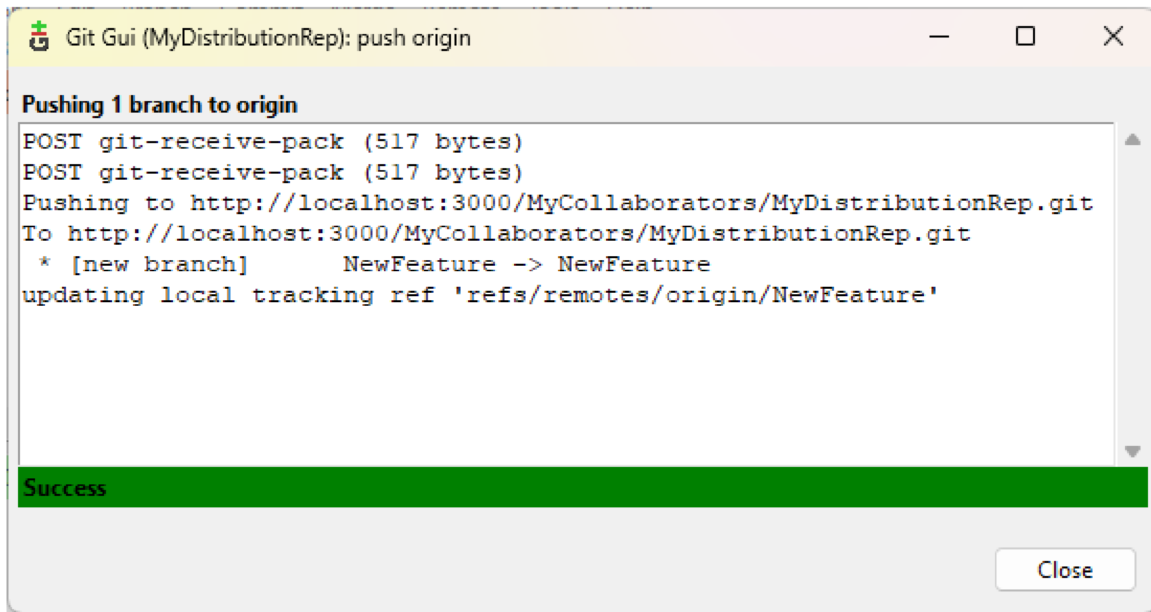


Figure 119: Push popup, suite.

The NewFeature branch pushed to the collaborative repository:

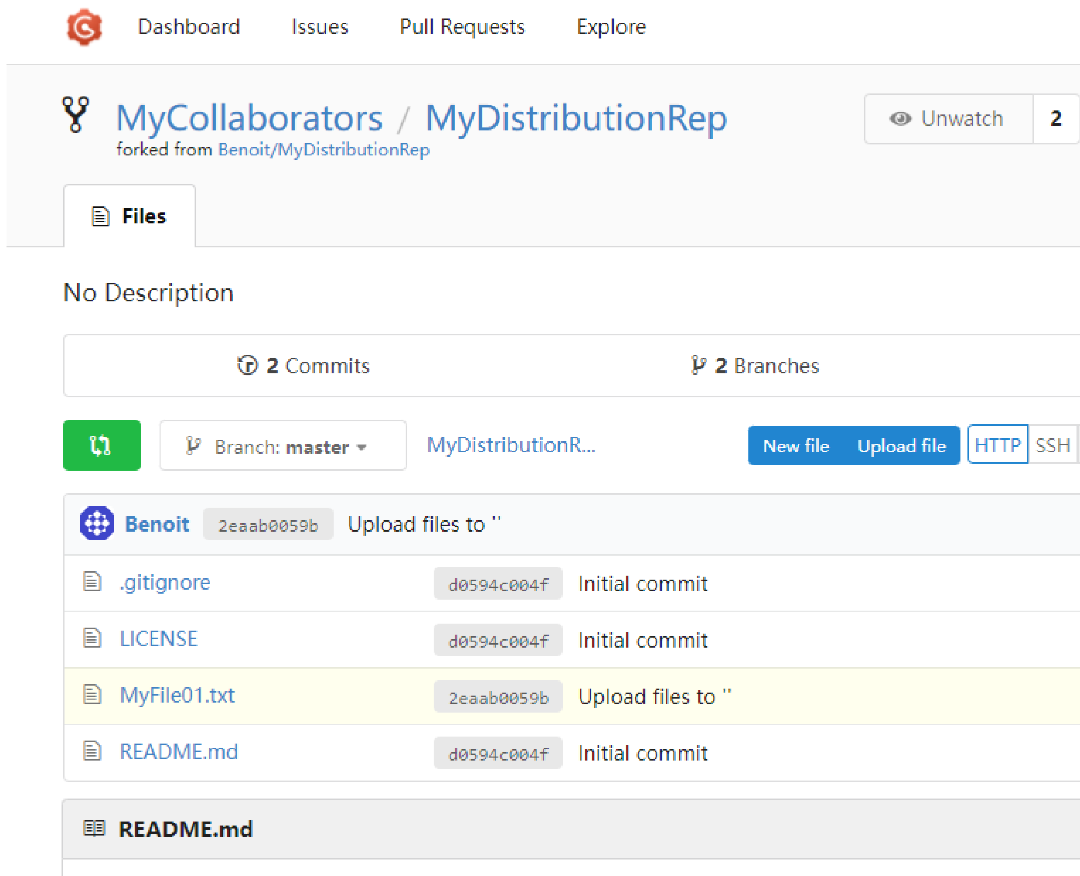


Figure 120: The NewFeature branch pushed to the collaborative repository.

Hit now the green compare button:

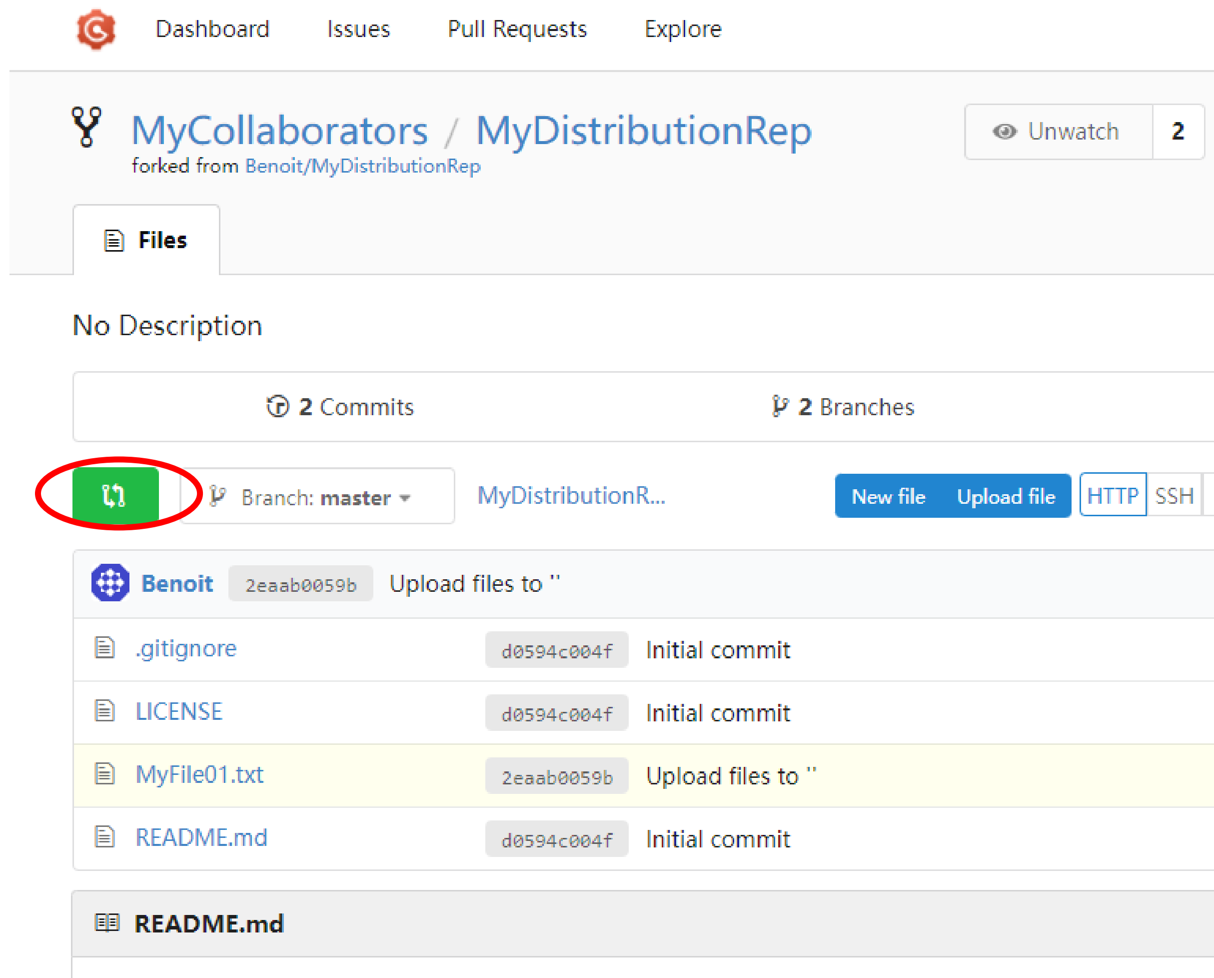


Figure 121: Comparison between the two branches.

Choose the NewFeature branch to make the comparison, and notice three things (see the next picture):

- The choice of the branch to compare to (red circled),
- The fact that you can add descriptions and attach files (green circled),
- The popup is now on the main repository (purple circled):

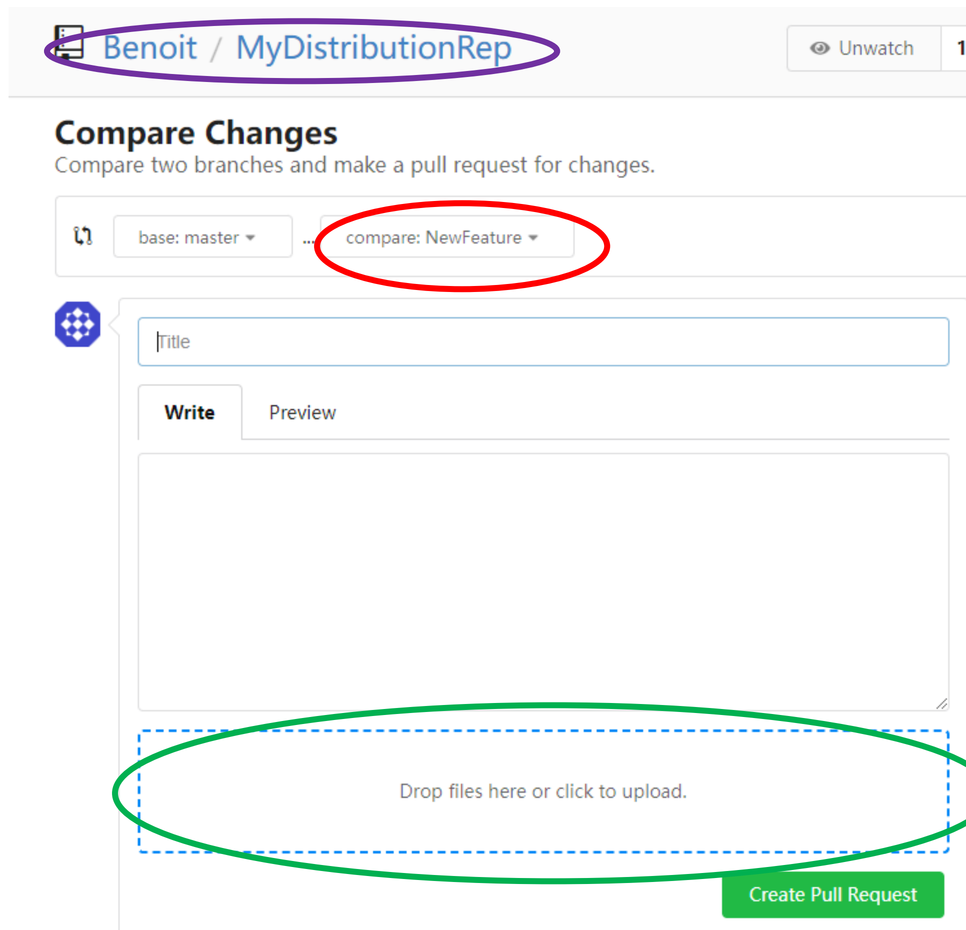


Figure 122: Pull request creation.

Assign someone to the review:

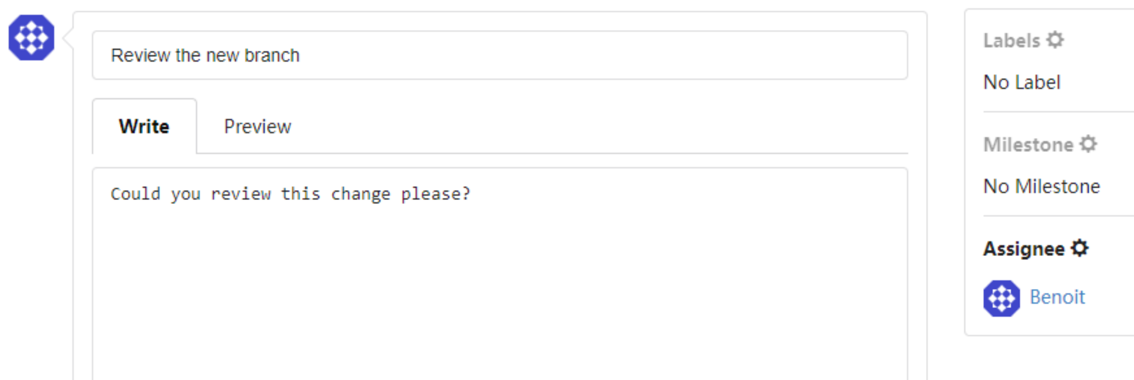


Figure 123: Benoit is assigned to the pull request review.

You can start a discussion or merge the pull request (red circled):

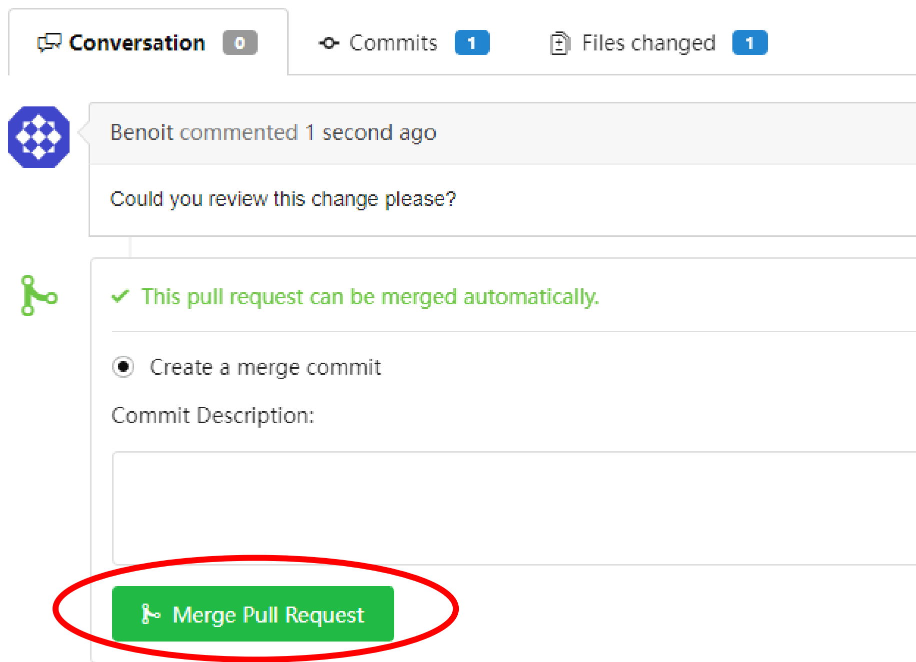


Figure 124: Merge pull request.

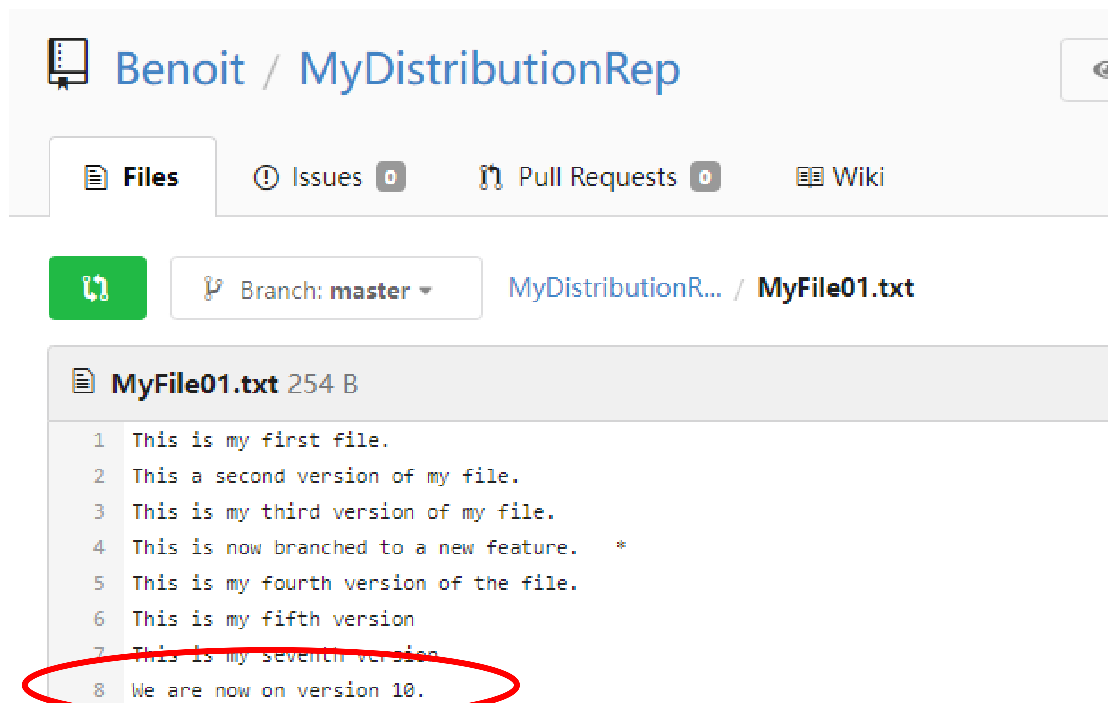


Figure 125: Merged change.

The MS Visual Studio client.

Create a ASimpleApp repository:

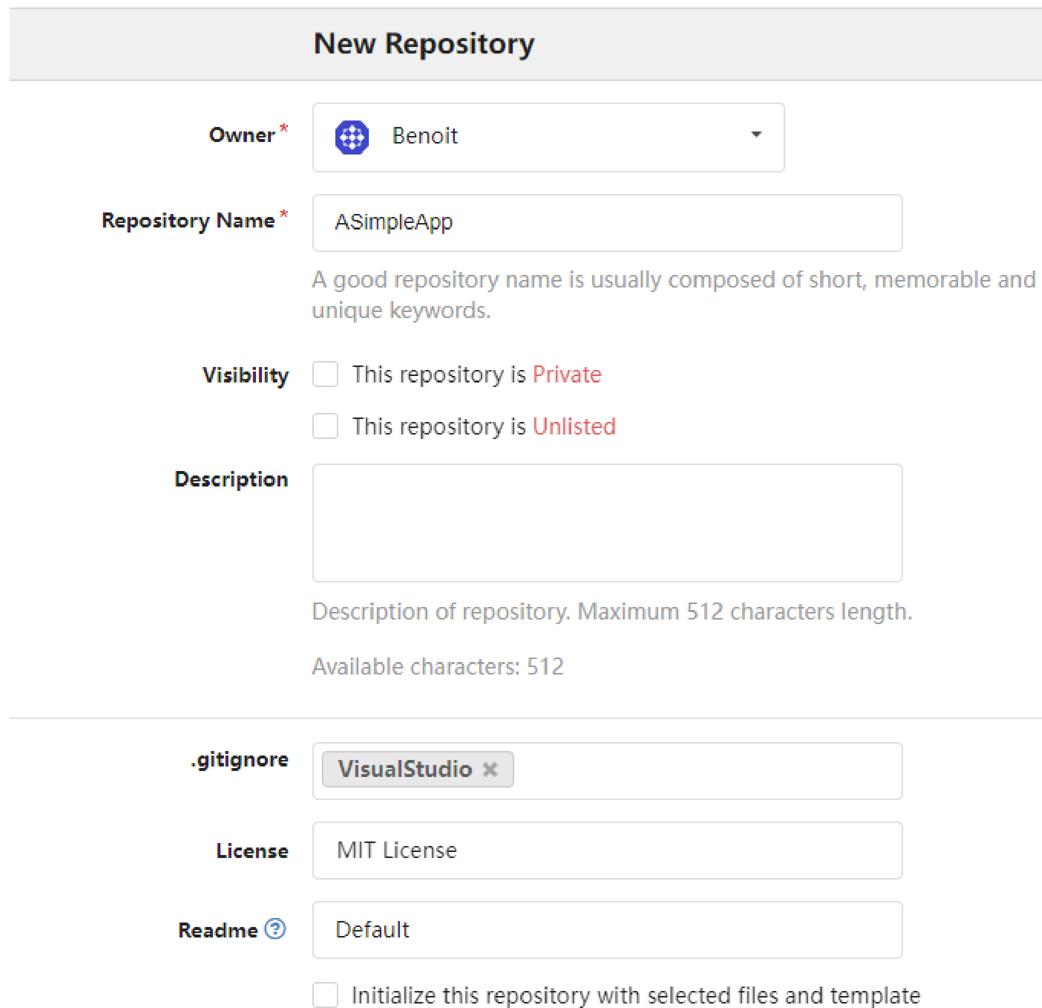


Figure 126: ASimpleApp repository.

Now launch MS Visual Studio and create a C# desktop application:

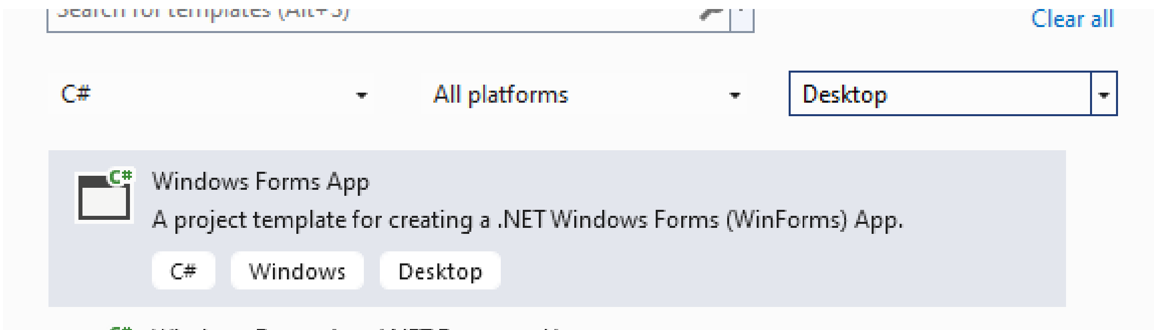


Figure 127: Create a C# desktop application.

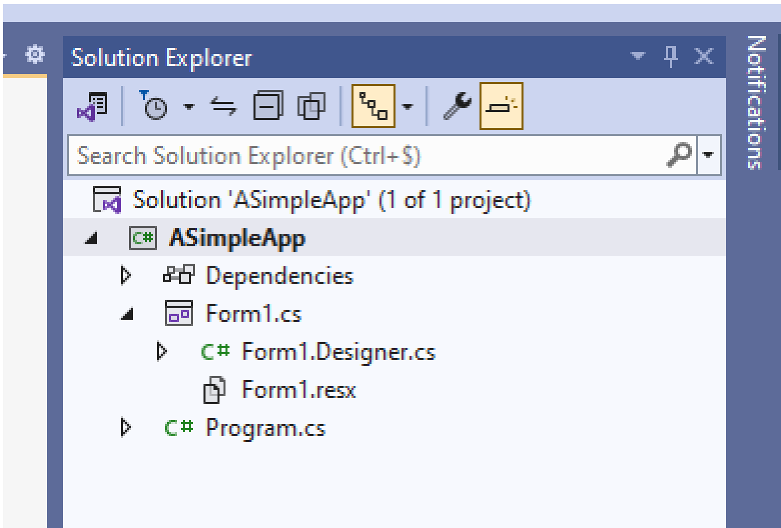


Figure 128: ASimpleApp C# application.

Create a local Git repository:

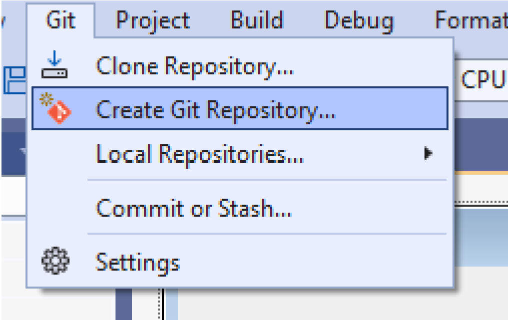


Figure 129: ASimpleApp Git repository.

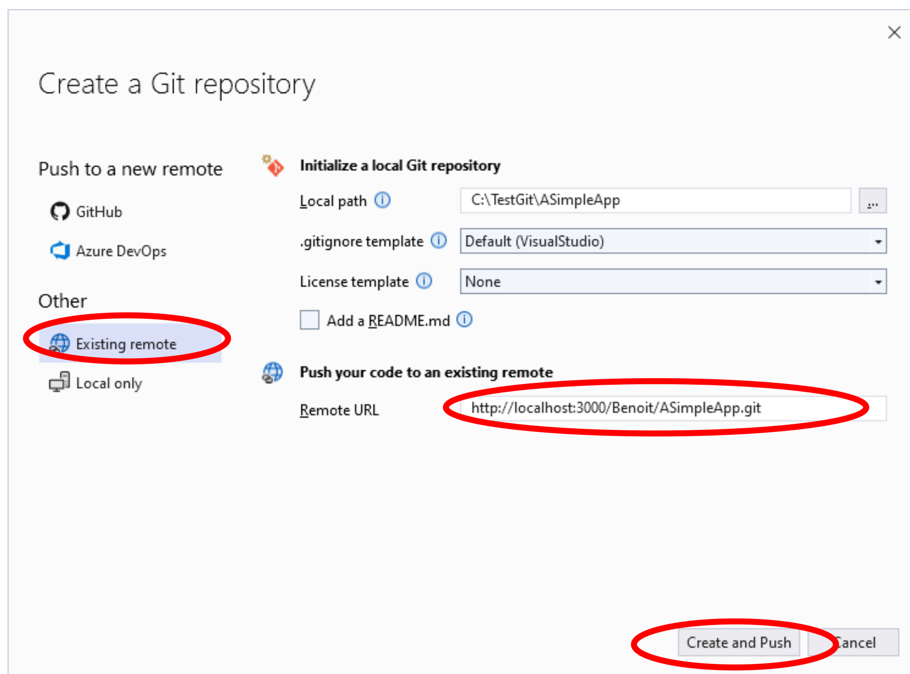


Figure 130: ASimpleApp local Git Repository.

Choose:

- Existing Remote
- The correct Remote URL (cut & paste from Gogs)
- Create and Push it.

N.B.

All Git operations are performed under an account. When the remote (*origin*) repository is hosted by GitHub or Azure DevOps is involved, the account is a GitHub or Azure DevOps one. In our case, since we use local Windows accounts, the accounts come in fact from the one defined in Gogs. We defined two accounts in Gogs, Benoit and Benoit02.

This can be configured in MS Visual Studio in the following way:

- In Git->Settings:

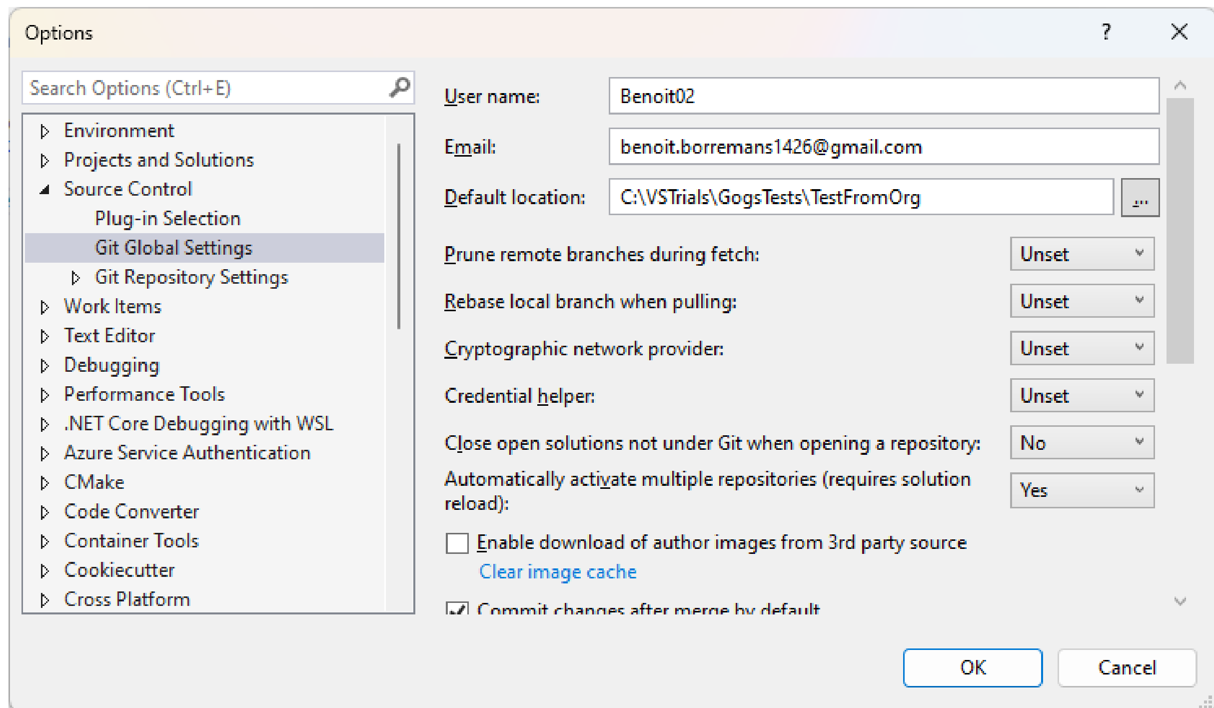


Figure 131: Git MS VS Settings.

Perform a Clear image cache.

- When you launch MS Visual Studio for the first time and perform an operation like a *clone* one, the account used will be the one that will be in the cache, that is the one used during the last git operation in MS Visual Studio.
- The settings defined for the project:

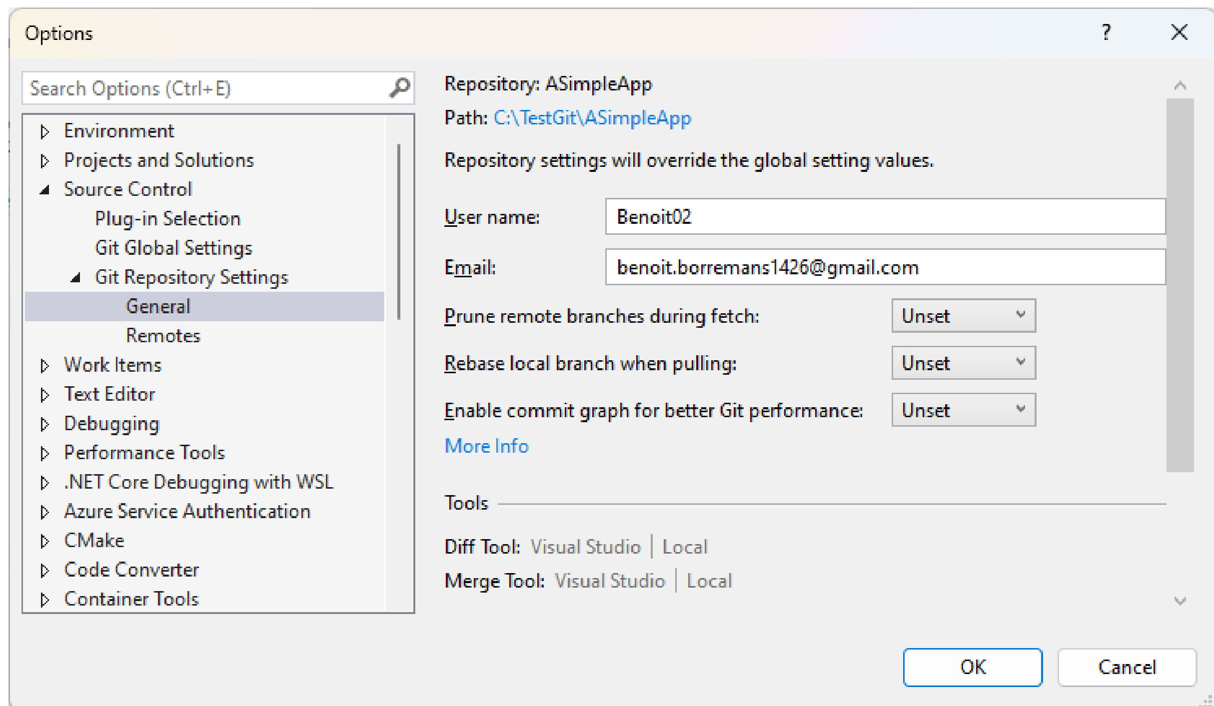


Figure 132: Project Git Settings.

Once pushed from MS Visual Studio, the Gogs repository will look like:

Benoit / ASimpleApp

Unwatch 1

Files | Issues 0 | Pull Requests 0 | Wiki

No Description

2 Commits | 1 Branches

Branch: master | ASimpleApp | New file | Upload file | HTTP | SSH | htt

Benoit02	b588e8c3cd	Add project files.
.gitattributes	2046c29351	Add .gitattributes, .gitignore, and LICENSE.txt.
.gitignore	2046c29351	Add .gitattributes, .gitignore, and LICENSE.txt.
ASimpleApp.csproj	b588e8c3cd	Add project files.
ASimpleApp.sln	b588e8c3cd	Add project files.
Form1.Designer.cs	b588e8c3cd	Add project files.
Form1.cs	b588e8c3cd	Add project files.
Form1.resx	b588e8c3cd	Add project files.
LICENSE.txt	2046c29351	Add .gitattributes, .gitignore, and LICENSE.txt.
Program.cs	b588e8c3cd	Add project files.

Figure 133: The pushed Gogs repository.

Once pushed, you can see three indicators in MS Visual Studio:

- In purple, the current branch,
- In green, the currently uncommitted changes,
- In red, the differences between the current repository and the remote (*origin*) one.

Let’s now add a button whose purpose is to say “Hello!”:

```
namespace ASimpleApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello!");
        }
    }
}
```

Figure 134: Say Hello button.

If you look at the indicators:

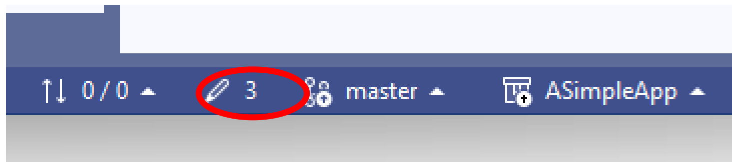


Figure 135: MS VS indicators.

You see that there are three uncommitted changes (The Initialize form, the button addition, and the message).

You can now commit the changes:

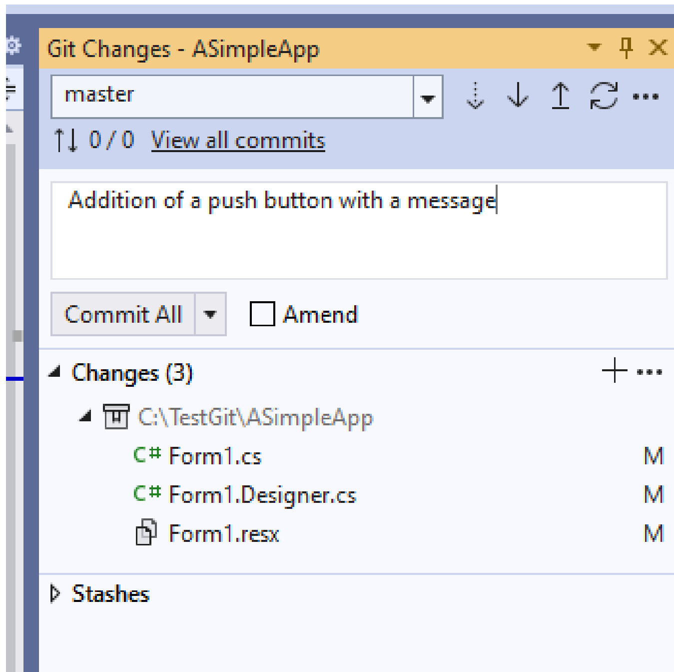


Figure 136: Changes committing.

Looking again to the indicators:

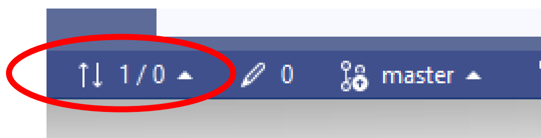
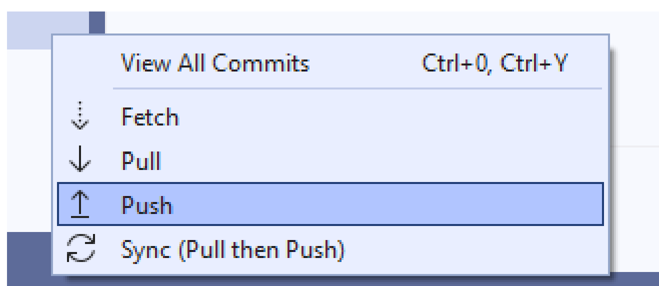


Figure 137: MS VS indicators.

You see that there is a difference between the local and the remote repositories.



You can push it.

Figure 138: Push the differences.

And see the Gogs repository content:

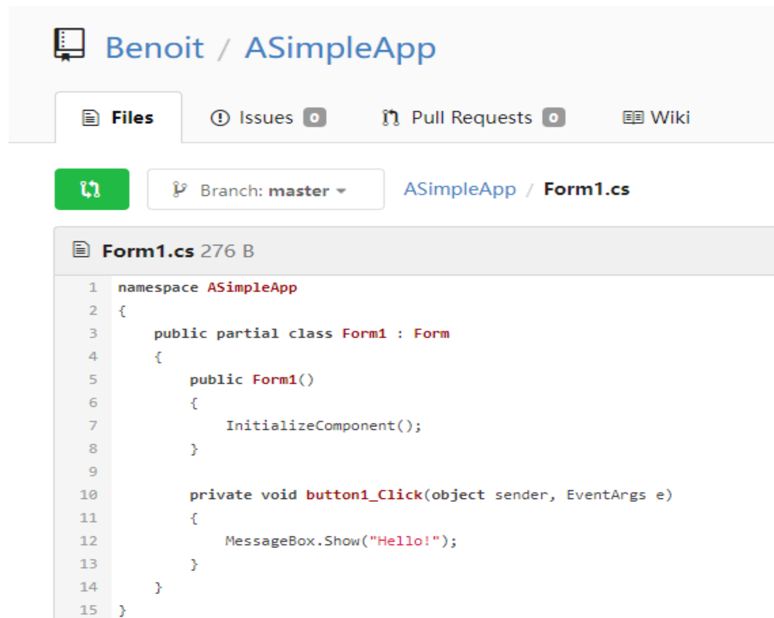


Figure 139: Gogs content.

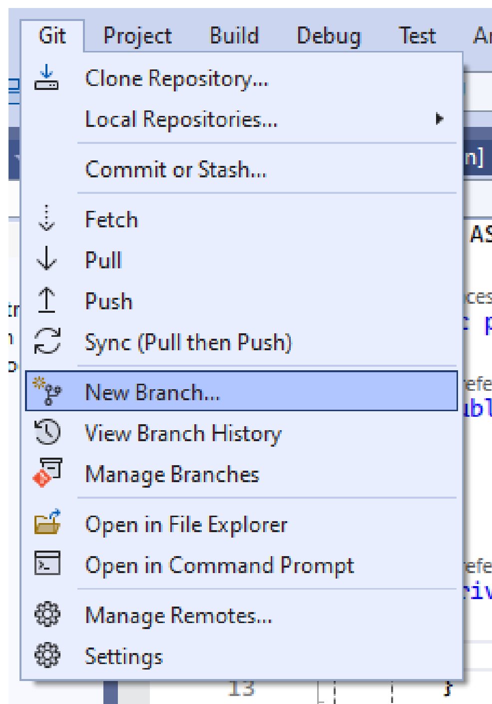


Figure 140: Create a branch from MS VS.

Let’s create a new branch.

And let's call it NewFeature:

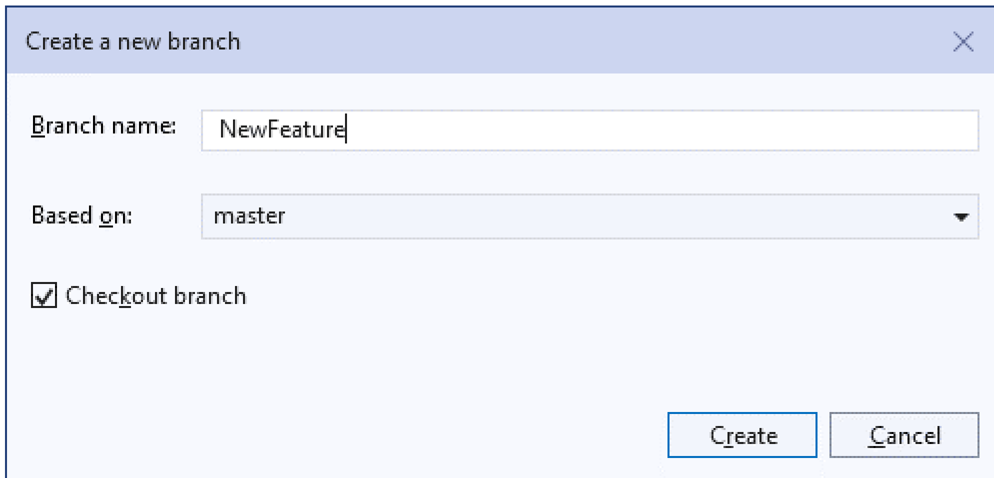


Figure 141: MS VS NewFeature branch.

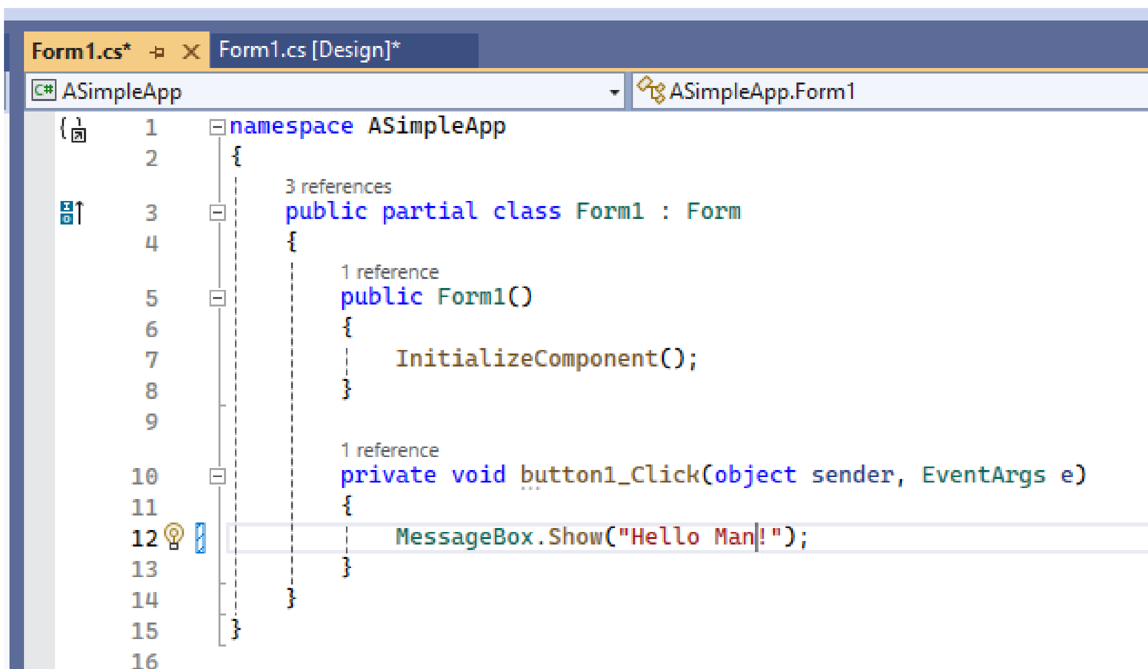
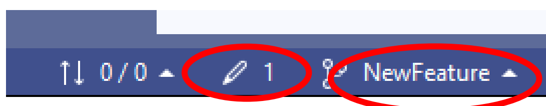


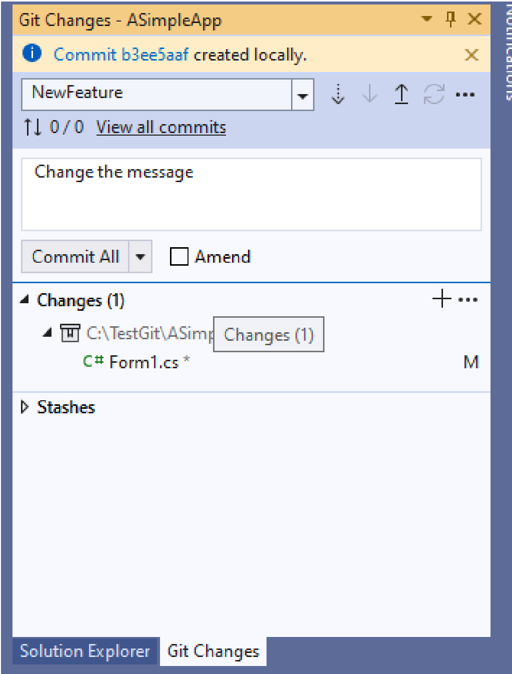
Figure 142: Change the message.

And change the message.



Again, looking to the indicators you see:

- That you are on the NewFeature branch,
- There is one uncommitted change.



Let's commit the changes.

Figure 143: Commit the changes.

And having a look to the indicators:

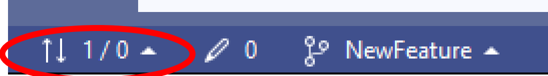
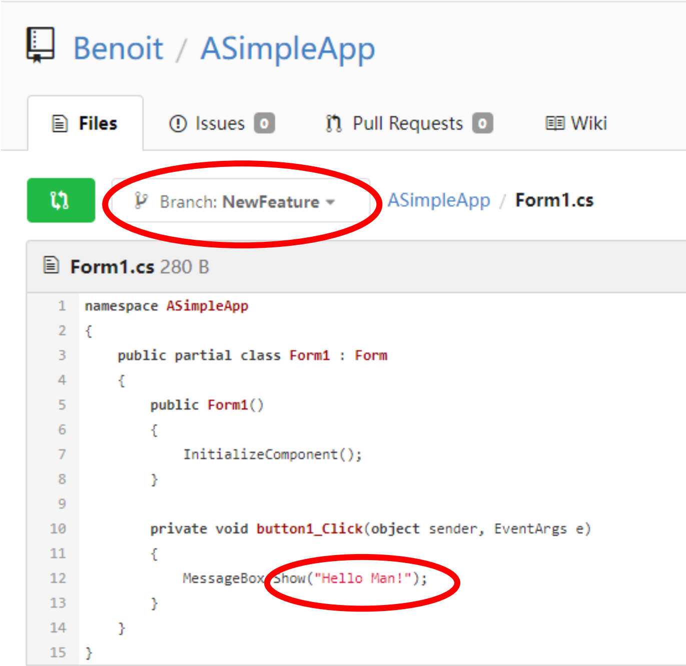


Figure 144: MS VS indicators.



You see the change on the NewFeature branch.

Figure 145: Changes in the NewFeature branch.

The git communication protocol

As we described it all in this document, the git transfer protocol is mainly designed to transfer files and their history from a collaborating repository to a main or another collaborating repository.

It is essential that this transfer is efficient, this is why, actually, the whole package files are not transferred, but rather only deltas between the files, together with information associated with the tags (the commits) and labels.

In order to make this transfer is consistent, when a branch is transferred from one repository to another, they must share the same history, the reason why, when a conflict occurs, the transmitted branch must be reconciliated with the target one on the remote.

Attention must be paid on large (binary) files, the reason why, for example, Gogs has a special treatment for those transfers.

On top of that, some communications between some hubs (hosts) and customers (clients) use additional protocol. This is the case between GitHub and MS Visual Studio when a push happens, in which case a pull request is automatically created – a point that is not addressed in this document.