TELEMAC-MASCARET SYSTEM

Git Guide

Version v8p3
December 6, 2021

Contents

1. Introduction

`Git` is a version control software that allows you to track changes in your project.

The principle is that you will tell `Git` to track changes in some files and take snapshots of these files throughout the evolution of the project by **committing** your changes. A **commit** is then a snapshot of a project at a given time, which tracks the changes made by one person on one or several files compared to the previous version. Every **commit** is a version of the project.

Here is an overview of all `Git` commands:
https://training.github.com/downloads/github-git-cheat-sheet.pdf

1.1 Required packages

First, to benefit from certain key features, you will need to install a recent version of Git (2.22.0 or over) as well as the Git LFS extension.

1.1.1 For Linux users

To install Git and Git LFS on Linux, you can do so through the package management tool that comes with your Linux distribution.

If you are on a Debian-based distribution, such as Ubuntu, try `apt`:

```
$ sudo apt install git-all
$ sudo apt install git-lfs
```

If you are on Fedora (or any closely-related RPM-based distribution, such as RHEL or CentOS), you can use `dnf`:

```
$ sudo dnf install git-all
$ sudo dnf install git-lfs
```

For more options, there are instructions on the official Git website for installing `Git` on several Linux distributions, at https://git-scm.com/download/linux.

1.1.2 For Windows users

You can download and install the latest “Git for Windows” from the official website at https://gitforwindows.org. Make sure Git LFS is checked within the Components page, and override the default branch name to main.

Other settings include the following, from which you need to choose:

- To use Git from the command line and also from 3rd party software (such as Git Extensions or other GUI client) and use the OpenSSL library;
- To checkout as-is (Linux) and commit Unix-style;
- To work directly within a DOS console windows;
- To please use the **git pull --rebase** option (see warning below for that command);
- To use the latest Git Credential Manager Core; and
- To enable file system caching.

1.2 GUI client

While it is better to learn Git through the command line, it is still recommended to use a GUI client, if only to have a better visibility of the commit history. `Git` already comes with a basic client called `gitk`, however it is best to use a more advanced one. Depending on your OS, we recommend one of the following:

- Linux: `SmartGit`, which is free of charge when working with open source projects and can be installed in your HOME directory, if you don't have administrator rights.
- Windows: although `SmartGit` can also be used, the recommended client is `Git Extensions`, which can be installed using a setup or as a portable application.

There are many other GUI clients, the most popular ones being `Sourcetree` (Windows only) and `GitKraken` (Windows, Linux and macOS), however, both require an online account to use them.

2. Using Git for TELEMAC-MASCARET

In this chapter, we will see how to start using Git for development with TELEMAC-MASCARET.

2.1 Cloning the repository

The TELEMAC-MASCARET source code is currently hosted on a GitLab server maintained by EDF R&D and available at `https://gitlab.pam-retd.fr/otm/telemac-mascaret`. You have to **clone** this Git repository, and by default the entire repository is cloned. Options to limit the size of the cloned repository on your local system are available and provided below.

2.1.1 Complete clone

From a terminal:

```
$ git clone https://gitlab.pam-retd.fr/otm/telemac-mascaret.git
  my_opentelemac
$ cd my_opentelemac
```

Here the clone will be made within the directory named **my_opentelemac** but you can name it as you wish.

2.1.2 Single branch clone

It is possible to clone a single branch as follows:

```
$ git clone -b feature_branch --single-branch
  https://gitlab.pam-retd.fr/otm/telemac-mascaret.git
  my_opentelemac/feature_branch
```

The disadvantage of this process is that you can no longer recover another branch within **my_opentelemac** without doing a low-level manipulation on the local repository. Therefore, cloning a single branch is not recommended, unless you work in a separate directory for each branch.

2.1.3 Shallow clone

It is also possible to make a “shallow clone” which consists in limiting the history to a certain number of commits. For example, to retrieve the last 20 commits only:

```
$ git clone --depth 20
  https://gitlab.pam-retd.fr/otm/telemac-mascaret.git
```

However, this method is not recommended either outside of a repository submodule or a continuous integration environment.

2.1.4 Partial clone

Finally, it is possible to make a "partial clone" by filtering the directories, which consists in recovering only part of the Git repository. This process requires a recent version of Git as well as the presence of a filter file on the system repository.

For TELEMAC-MASCARET, such file has been included on the repository that ignores the examples. Other filter files can be added upon request. The Git command is as follows:

```
$ git clone --sparse --filter=sparse:oid=main:.gitfilterspec
  https://gitlab.pam-retd.fr/otm/telemac-mascaret.git my_opentelemac
$ cd my_opentelemac
$ git config --local core.sparsecheckout true
$ git show origin/main:.gitfilterspec >> .git/info/sparse-checkout
$ git checkout main
```

The above list of commands is quite heavy as Git is not at all intended for partial clone, but it works.

2.2 Cloning the repository – GUI users

All GUI clients provide a clone command located under the main menu, for example in "Repository > Clone" for SmartGit, or "Start > Clone Repository..." for Git Extensions.

Most interfaces provide you with direct access to single branch and/or shallow clone. You can do a partial clone using the command line description above.

2.3 Configuring developer credentials

People who requested a developer access should all have received an e-mail to access the GitLab server when their account was created.

If the identifiers are not necessary to clone the repository, since it is public, it is obviously not the same to modify it. Access rights are required through a two-factor authentication (2FA). Both authentication factors are required every time you log on to GitLab.

The first level authentication factor is your GitLab username and password. The second level factor is a code that is generated through an OTP (One-Time Password) application that you have to install on another device, such as your phone. Such applications include FreeOTP or Microsoft Authenticator, but many others are available for Android and iOS.

Fortunately, this OTP is only necessary when accessing the GitLab web page. But when working with Git, the access is stored locally on your system in the form of a personal **access token**.

To generate this token, you need first to connect to the GitLab web interface. On that occasion, you will need your GitLab username and password as well as the OTP code. Once logged on, the access token is available through **Edit profile**, selecting **Access Tokens**.

You must then generate the access token by filling in a name (of the application that uses Git, for instance) and making sure that you have checked **read_repository** and **write_repository** at least.

Then click on **Create personal access token** for the key to appear at the top of the page: note it down, copy/paste it, keep it, as you will lose it once you leave the web page.

2.3.1 Linux users

In order to avoid entering the token each time you push to the repository, you need to enter the following command to store your credentials:

```
$ git config --global --credential.helper store
```

Then, the token is to be written to $HOME/.git-credentials in the following form:
`https://oauth2:T<TOKEN>@gitlab.pam-retd.fr`
Of course, you will need to replace **<TOKEN>** with your own token.

From here you can work.

2.3.2 Windows users

Since `Git` **Credential Manager Core** should have been enabled through the `Git` setup, storing your credentials requires only to enter the token once and for all, for example when cloning the repository (or later on, when doing a push):

```
$ git clone
  https://oauth2:<TOKEN>@gitlab.pam-retd.fr/otm/telemac-mascaret.git
  my_opentelemac
```

2.4 Configuring the repository

Before to start working with `Git`, you need to indicate the username and e-mail address that will be associated with the commits.

For TELEMAC-MASCARET, we recommend using your first and last name, by entering, from the repository directory:

```
$ git config user.name "FirstName LastName"
$ git config user.email firstname.lastname@mycompany.com
```

You can also configure `Git` to use the text editor of your choice for editing commit messages. Otherwise, the default `Git` interface will be used. To set it (.e.g to use `Emacs`), enter:

```
$ git config core.editor emacs
```

To use the same settings for all your `Git` repositories, you need to pass the **global** option to the above commands, e.g.:

```
$ git config --global core.editor emacs
```

2.4.1 Proxy settings

If your organization is behind a proxy you need to tell Git to use it as below:

```
$ git config --global
  http.proxyhttps://<username>:<password>@<proxy_address>:<proxy_port>
$ git config --global
  https.proxy https://<username>:<password>@<proxy_address>:<proxy_port>
```

By replacing:

- <proxy_address>: by the address to your proxy;
- <proxy_port>: by the port of your proxy;
- <username>: by the login to your proxy, if any;
- <password>: by the password to your proxy, if any.

3. Working on branches with Git

3.1 Creating a branch

Git saves snapshots of your project when you make commits, and **always** attributes them to the **branch** on which you are located. A branch is composed by a "homogeneous" set of commits: for example, you can have a branch containing all your stable version project, a branch containing the future stable version, development branches where you introduce a new feature in the code, etc.

The branch **main** is created by default. We usually use it for stable versions of the code.

To add a new branch, enter:

```
$ git branch my_branch
```

To remove a branch, enter:

```
$ git branch -d my_branch
```

To tell Git you want to work on a specific branch, enter:

```
$ git checkout my_branch
```

Then all the subsequent commits will be done in **my_branch**.

3.2 Developing on branches

Once you have told Git to position you on a branch, you can start to work on it by modifying the existing files or by adding new ones.

Important: before committing, you need to specify which files will be committed by adding them to the **stage area**. Files which are not added to this space will not be part of the next commit.

Therefore, to include a modified or created file in the next commit, you need to enter:

```
$ git add filename.ext
```

You can also choose to only add part of the modifications that you have made to a file, using the **-p** option:

```
$ git add -p filename.ext
```

Then Git asks you directly from the terminal if you want to stage (≡ to add) each part of your modifications (Git identifies them as blocks of modifications). For each part that you want to add, type **y** and enter, otherwise **n** and enter.

To remove a file versioned from the repository, you need to enter:

```
$ git rm filename.ext
```

To remove files from the **stage area**, e.g. if you added a file by mistake, enter:

```
$ git reset -- filename.ext
```

Once you have added and/or removed all the files you wanted, you can commit them:

```
$ git commit -m "My commit message"
```

The commit message should describe the changes and should not exceed 72 characters.

You can also enter a more detailed message by using the text editor. To do so, do not specify a message when calling the **commit** command, as below:

```
$ git commit
```

Git then pops up a text editor window and asks you to enter a **commit message**. Enter the message, then save and close the editor.

After a commit, Git tells you a new commit has been made on the branch:

```
[main 537e9e633] My commit message
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 sources/telemac2d/my_new_file.f
```

Note: you can choose to commit all the modifications that have been made in the currently tracked files by doing as follows:

```
git commit -a "My commit message"
```

However, this command will not include new files.

3.2.1 Commit messages

Commit messages are quite important so here are some tips to make them better: http://robots.thoughtbot.com/5-useful-tips-for-a-better-commit-message

You should structure your commit message like this (from http://git-scm.com/book/ch5-2.html):

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Further paragraphs come after blank lines.

 \- Bullet points are okay, too

 \- Typically a hyphen or asterisk is used for the bullet, preceded by a
   single space, with blank lines in between, but conventions vary here
```

3.2.2 Best practices

Commit your developments very regularly.

For each sub-development, a commit should be done: when you create a new function, a new variable, a new class, when you fix a bug, etc.

Do not hesitate to do a lot of commits: once your development are finished, you can squash them to fewer and larger commits for better readability of the commit history (see below).

A commit may only contain modifications of a few lines in one file as long as that modification has a significant impact.

One useful thing is that you can navigate in the branch to find out where a bug was introduced (making a dichotomy is usually fast). Therefore, it is important that **for every commit you make, the code has to compile without errors**. Otherwise, bug tracking is made very difficult.

3.3 Useful commands

The command:

```
$ git log
```

shows the history of your branches, where you can see that an ID was assigned to each commit. This ID is SHA-1 hash of every important thing about the commit. You can also get commit IDs from your GUI client, such as `gitk`.

You can revert the modifications introduced in a commit by entering:

```
$ git revert commit_id
```

You can see the differences in a file between the current and a previous version by entering:

```
git difftool commit_id filename.ext
```

where the diff tool may be `vimdiff` or `meld`, for instance.

To set the tool used by git type:

```
git config diff.tool meld
```

3.4 Move inside or in between branches

You can always come back to any version of your project by coming back to the corresponding commit.

To move to a given branch:

```
$ git checkout branch_name
```

To position the repository at a given commit:

```
$ git checkout commit_id
```

`Git` will usually refuse to move if you have uncommitted changes in your project. To see the uncommitted changes, enter:

```
$ git status
```

If you want to **move and erase the uncommitted changes** (which may be quite risky):

```
$ git checkout -f commit_id
```

If you want to temporarily save your changes to be able to recover them later:

```
$ git stash
```

The stash is a space for saving temporary changes that you may want to apply later.

You may now move around in the history unhindered. You may have several sets of modifications in the stash, you can list them by entering:

```
$ git stash list
```

Each stash has a number assigned to it (stash{0}, stash{1}, etc.).

Apply the changes contained in one of the stashes (here the number 3):

```
$ git stash apply stash{3}
```

Entering only **git stash apply** will apply the changes contained in stash{0}.

Erase the content of given stash (here number 3):

```
$ git stash drop stash{3}
```

Entering only **git stash drop** will drop the changes contained in stash{0}.

Apply and erase the changes contained in one of the stashes (here number 3):

```
$ git stash pop stash{3}
```

Entering only **git stash pop** will apply and erase the changes contained in stash{0}.

3.5 Merge developments in between branches

A very interesting feature of Git is that it allows you to apply the changes made in one branch to another one quite easily. There are two commands for this, namely **git merge** and **git rebase**. Here we will focus on rebasing. The two techniques are quite equivalent, but rebasing is considered cleaner and as such, is used as a coding convention in TELEMAC-MASCARET.

Let's say you have created a branch called **my_branch** that you have used to develop a new feature. In the meanwhile, someone has made developments in the branch **other_branch**, and you now want to apply your development to **other_branch**.

First, if **other_branch** is a shared branch, you may not want to share all the commits you have done with such a degree of detail as for your personal use. Therefore, it is recommended to follow the process below:

- Backup your branch:

  ```
  $ git branch my_branch_backup
  ```

- Modify the commit history by using an interactive rebase:

  ```
  $ git rebase -i HEAD~5
  ```

This command will prompt the text editor where the 5 latest commits (any number works of course) in the branch are displayed. You can then decide to squash some of them (put them together), reword the commit messages, edit the content of the commits.

The way you can do it is explained in the text file that Git prompted for the interactive rebase. If you change the position of the commit in the column, it will modify the order of the commit in the branch accordingly. Deleting a commit in the text editor also deletes it in your branch.

Note that giving a commit ID for the interactive rebase also works:

```
$ git rebase -i commit_id
```

- Once you have cleaned the branch as you wish, rebase it on top of **other_branch**:

  ```
  $ git rebase other_branch
  ```

  This will move the branch **my_branch** on top of **other_branch**.

  In case the same file was modified at the same place in the two branches, Git won't be able to automatically do the rebase and will notify a conflict in the file. It will stop the rebase and edit the conflictual files, leaving both versions unmodified to let you choose what you want to keep. Open the files in a text editor and modify them to remove all the conflicts. **Check that the code correctly compiles after your modifications.**

  After the files have been edited, you need to add them:

  ```
  $ git add conflictual_file_1 conflictual_file_2 ...
  ```

  Then you can continue the rebase:

  ```
  $ git rebase --continue
  ```

  In case you don't want to carry on with the rebase, enter:

  ```
  $ git rebase --abort
  ```

- Once the rebase is done, and you have checked that everything works as expected (compilation, test-cases), update **other_branch** and remove **my_branch**:

  ```
  $ git checkout other_branch
  $ git merge my_branch
  $ git branch -d my_branch
  ```

4. TELEMAC-MASCARET Git workflow

In conclusion of this document, the TELEMAC-MASCARET Git development workflow is presented.

- First make sure all your shared branches are up-to-date with origin:

  ```
  $ git pull --rebase
  ```

- Create a new local branch and start working on it (make one branch for each new feature you plan to introduce in the code):

  ```
  $ git checkout -b my_branch
  ```

- Push your branch to the server:

  ```
  $ git push --set-upstream origin my_branch
  ```

- Often commit, making small commits containing specific change at a time:

  ```
  $ git add modified_file_1.ext folder/modified_file_2.ext
  $ git commit -m "My commit message"
  ```

  Note: use **git add -p** if necessary.

  Before committing, ensure that you have documented all your changes and that you respect TELEMAC-MASCARET coding conventions.

  All the commits must compile for debugging purposes.

- Once your developments are finished, you need to merge your branch with **main**. First, backup your branch:

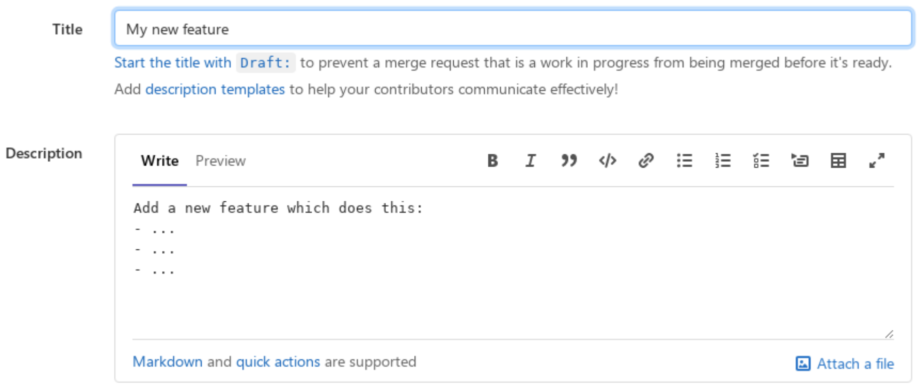
  ```
  $ git branch my_branch_backup
  ```

- Clean up your branch using an interactive rebase on the necessary amount of commits (*e.g.* 10):

  ```
  $ git rebase -i HEAD~10
  ```

Follow the text editor instructions to clean your branch history as needed. The idea is that **the history should be kept as clean as possible, so that it is easily understood by other developers**.

- Once you have finished cleaning the branch history, create a Merge Request (MR) by entering the following URL on your web browser:
  `https://gitlab.pam-retd.fr/otm/test-telemac/-/merge_requests/new?merge_request%5Bsource_branch%5D=my_branch`
  From the web page, enter a title, describe precisely what your branch does and create the MR, as in fig. 4.1.


Title: My new feature

Start the title with `Draft:` to prevent a merge request that is a work in progress from being merged before it's ready.

Add description templates to help your contributors communicate effectively!

Description:

```
Add a new feature which does this:
- ...
- ...
- ...
```

Markdown and quick actions are supported

Figure 4.1: Creation of a Merge Request

- If your branch cannot be merged, `GitLab` will disable the **Merge** button and tell you that there are conflicts, as shown in fig. 4.2.

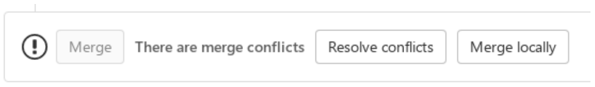


Figure 4.2: Conflicts in a Merge Request

To fix those conflicts, rebase **my_branch** on top of **main**:

```
$ git rebase main
```

- Solve each conflict `Git` encountered during the rebase and **make sure that the code compiles before continuing the rebase**.

- Once the rebase is finished and that **you have verified that the test-cases run correctly**, force push your branch to update your **MR**:

```
$ git push --force-with-lease
```

Important: you should never use the **--force/-f** option as other developers may have pushed new commits to the branch while you were rebasing it. Using **--force-with-lease** will prevent the update of a branch in such a case.

- After conflicts have been solved or if there wasn't any, the **MR** can be accepted. If you have **Maintainer** rights on `GitLab`, you can merge it yourself. However, if the changes brought by the **MR** are quite important, you should ask another developer to review your changes, a task which can be done from the **MR** itself.

If you don't have **Maintainer** rights, you will need to ask for a **Maintainer** to review your code changes. Maintainers can be found from the **Project members** list which is available from the GitLab project webpage.

- After the **MR** has been accepted, delete your branch if needed and also delete your backup branch:

```
$ git branch -d my_branch
$ git push origin --delete my_branch
$ git branch -d my_branch_backup
```