Git – First Steps

Supplementary Material

Mateusz Szumilas, PhD Eng.

1. Introduction

Version control allows us to keep track of changes made to the files. The set of files and a history of their changes are called a *repository* (*repo*). Such a repository may be created and managed using dedicated systems, one of them being Git. Git is a distributed version control (DVC) system, where the repository is copied to the user's local storage. That is, not only the newest versions of files, but also their complete history of modifications, are available locally. An alternative to the DVC systems is centralized systems, based on a remote, central storage that keeps all the files' versions.

To work with Git, one should start creating a new repo or copying an existing one, then proceed with applying changes and (optionally) sending them to the remote repository. These registered changes are called a *commit*. An important feature of the Git and many other version control systems alike is the possibility of creating branches. A branch allows users to work with a code copy outside of the production code. When some (or all) of the introduced changes should be moved to the production, i.e., the master branch of the software, then a process called *merging* is executed. During merging, some conflicts may arise and should be solved. Such conflicts happen due to the changes introduced in the production version since the separation of the merged branch.

i

An official Git distribution may be downloaded from git-scm.com .

i

When a Git GUI is referenced in this document, it should be understood as one of the user interfaces installed with the Git official distribution, namely: `gitk` (a graphical repository browser) and `git-gui` (a tool for preparing and registering commits).

i

For development environments based on the Eclipse IDE an EGit, the Git integration for Eclipse, is available. It may be installed from the Eclipse Marketplace.

2. Creating a repository

There are two ways of creating a local repository:

1. by initializing a new repository in an existing directory,
2. by cloning a remote repository.

These methods are shortly described in the following subsections.

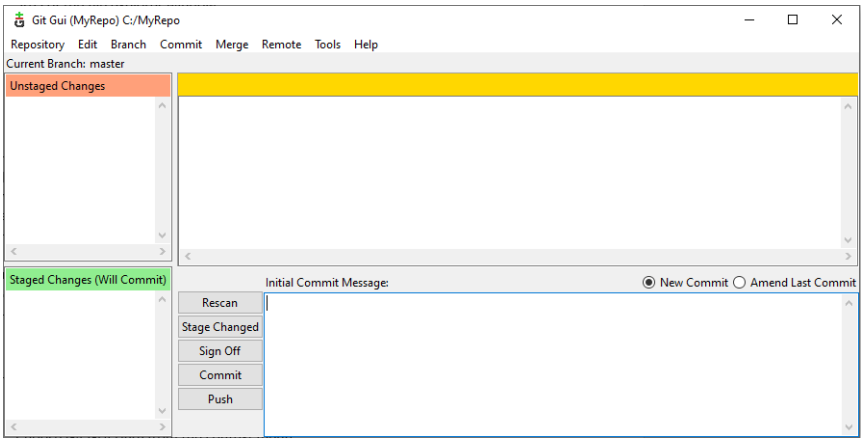
2.1. New repository

To initialize a new repository in the existing project directory (this directory may be already populated with files), you may use the Git GUI interface.

1. Right-click on the directory where you want to initialize the repository or in the empty area of the file explorer window.
2. Choose *Git GUI Here* from the context menu.
3. In the dialog window, choose *Create New Repository*.
4. Specify the path of the project directory (if it is the directory in which the *Git GUI* was opened, it is sufficient to specify "./") and accept.
5. The repository is now being initialized.

> **i** All the essential information for version control is stored in the *.git* subdirectory (hidden by default). From the internals of the *.git* directory, all the tracked files may be recovered.

When the repository is created, a graphical interface to Git opens automatically:



This window may be opened later by right-clicking on/in the repository directory and choosing *Git GUI Here* option from the context menu.

2.2. Cloning a repository

You may clone an existing repository with the Git GUI:

1. Right-click on a directory or in the empty area of the file explorer window.
2. Choose *Git GUI Here* from the context menu.
3. In the dialog window, choose *Clone Existing Repository*.
4. Specify the *Source Location*, for example, as SSH (starts with *git@*) or HTTPS address. Other possibilities are a local protocol and the Git protocol. If you provide the SSH URL, you probably need to have your SSH key generated to establish a secure connection (sometimes anonymous access through SSH is accepted). If you specify the HTTPS URL, you may clone the repository with or without authorization, and it depends on whether the repository is private or public, respectively.
5. As the *Target Directory*, specify the path to a non-existing directory. The directory is created during cloning.
6. Accept your input and start cloning.

The connection to the repository is remembered automatically and saved as a remote called "origin". The working copy of the last version of files is checked out, that is, the files are placed in the repository's directory. To get a new version of the remote repository, use *git fetch*. Fetching doesn't change the working directory, i.e., you get the current state of the repository but your working directory remains intact. After fetching, you may merge new changes. If you would like to combine the fetch operation with an attempt to merge automatically, use the *git pull* command instead.

If you work with a remote repository, you should fetch the changes often to keep track of its latest state.

3. Introducing changes

All the modified files are shown in the Git GUI section named *Unstaged Changes*. You can click a file name in this list to view changes found by Git. If new changes are made, the repository directory should be refreshed with *Rescan* button (or by pressing the F5 key). Recording your changes to the repository is named *committing*.

To register your changes in the repository:

1. Choose the files, which modifications you want to register, and move them to the *Staged Changes* group. To move the files:
   - left-click the file icon

     or
   - select one or more files and press Ctrl+T (add changes to the commit) or Ctrl+U (remove changes from the commit).

i	The files committed to the repository are *tracked*. All other files in the project directory are *untracked*.

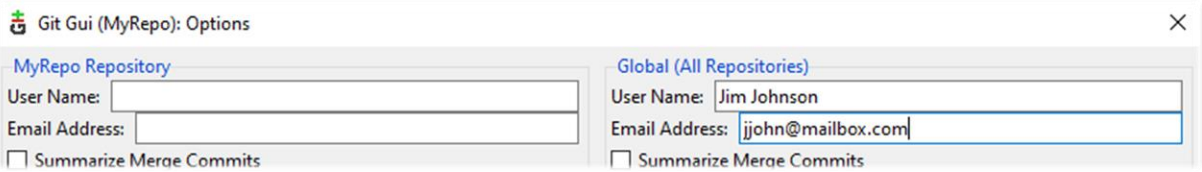
2. Add a description in the *commit message* field.

i	Remember always to add a short description of your changes: what and why was changed? Don't commit without a message or with a meaningless one.

3. Record your changes by pressing the *Commit* button.

i	It is a good practice to separately commit aesthetic changes in the code from the changes of its operation.

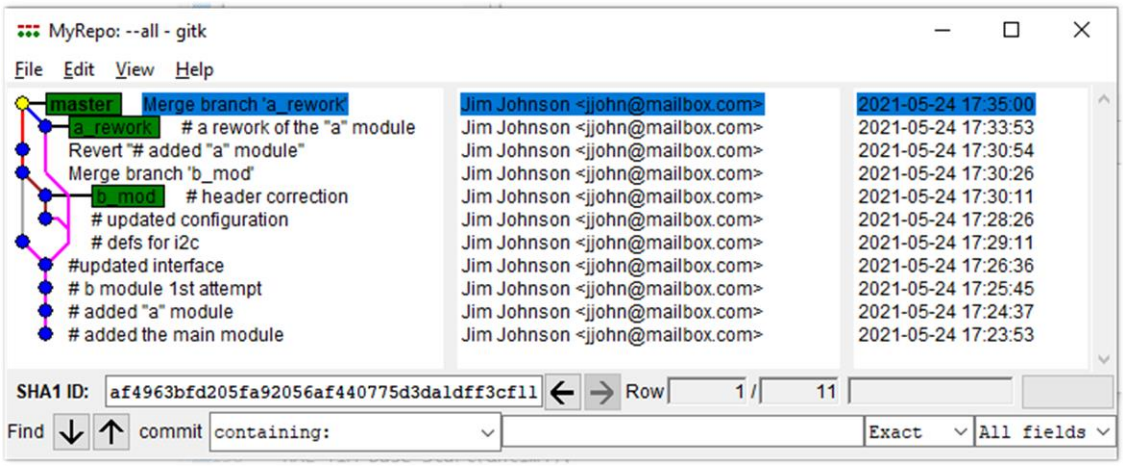
If it is your first commit after installing the Git, information about missing user data is displayed. In such a case, you should choose **Edit >> Options** from the menu and specify the user's name and e-mail address for current or for all repositories:



To send your commits to the remote repository, use the *git push* command (in the Git GUI: press the *Push* button or use Ctrl+P key combination).

!	An important rule: please remember NOT to push your work to the remote repository until you are fully satisfied with the results. Up to this moment, change the files locally.

The history of current or all branches may be visualized with a tree-like graph using the *gitk* application. To open it, choose the **Repository >> Visualize … History** option. An example of a history of a repository with three branches (*master*, *b_mod*, and *a_rework*), as visualized with the *gitk*:



4. Undoing changes

There are multiple ways of undoing changes in the repository with Git. However, most of them should be completed **locally**, as modifying the history of the remote repository should be avoided.

> **!** It is generally advised not to modify the history of public/shared/stable branches. Therefore, the changes in the history introduced by pushing to these should be treated as final.
> The history modifications should be limited to personal/local branches.

The changes may be introduced at different levels. First, try to understand the differences between the following three elements of the Git environment:

- working directory (tree) – the directory tree of files you are currently working with
- index (a staging area) – a single large binary file where Git prepares the next commit; it lists all files staged in the current branch
- HEAD – the pointer to the current branch reference, effectively to the last commit on the current branch

When you want to undo some of your changes, you may consider one of the following solutions (these are only examples of usage; these commands may also be used in other scenarios).

→ amend commit

You committed too early and forgot to add some files? You want to change your commit message? With *git commit --amend,* you may redo your commit, effectively overwriting it with its new version. Only amend a **local** commit that was not pushed anywhere to avoid causing problems for other repository users.

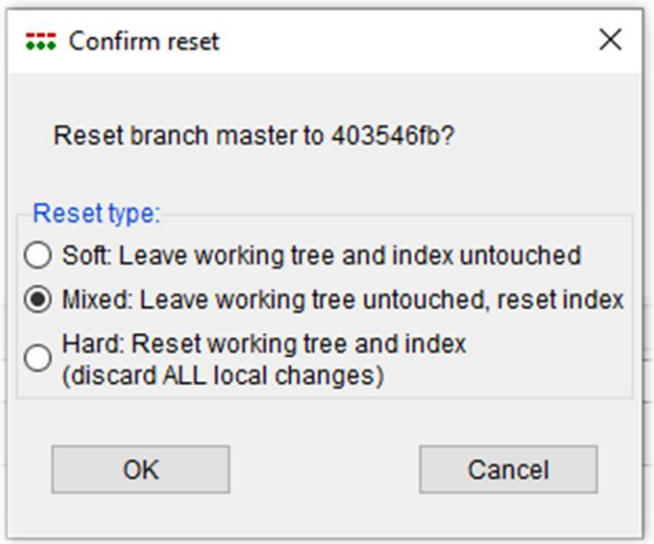
→ checkout commit

If you don't want to keep your uncommitted changes and discard them: checkout some valid commit, for example, the last commit, with which you have started introducing your changes.

Be careful, as you may lose some of your work.

→ reset commit

You committed changes **locally,** but now you would like to remove it from the repo's history? Use *git reset* to modify your branch. There are three options for this command; you have to decide which one should be used, for example, after choosing to reset in *gitk* repo's tree:



As described in the dialog:

- *git reset --soft* : changes the commit history (HEAD is moved, commit is removed), but the index and working directory remains unchanged;
- *git reset --mixed* : changes the commit history and additionally resets the index, that is unstages everything; the working directory remains unchanged;
- *git reset --hard* : be careful, as you may lose some of your work with this command; everything is reset to the state where the HEAD points to.

→ revert commit

If you want to undo some changes you have made and committed previously, you may revert it. The *revert* operation creates a new commit that undoes changes from the specified old commit. This allows for **keeping the history** of changes because the old commit is left intact.

5. Common problems

P1. There are some changes I would like to hide from working tree, but I don't want to discard them

Solution:

Separate the changes from the current branch by creating a new branch and committing your changes there. Then you can checkout the previous branch (e.g., **Branch >> Checkout…** in the Git GUI menu), where your changes are not available, and switch back to the newly created branch to continue your work.

P2.1. I can't merge the changes from the remote repository (*pull*) due to the uncommitted changes

P2.2. I want to put aside my changes for a while to make some other changes to the current (or some other) branch

Solution 1:

Check the solution of the P1 problem – commit your changes in a new branch.

Solution 2:

Use the `stash` command, especially when there is a need for a fast switch to another commit, and the changes should not be committed at this point.

To save away your local working directory state, use some of the following commands:

> **i** You may open the Git console, for example, using the **Repository >> Git Bash** in the Git GUI menu.

`$ git stash` or `$ git stash push`
tracked files with uncommitted changes are saved away; the working directory and index are rolled back to HEAD

`$ git stash push -m „message"`
tracked files with uncommitted changes are saved away with an additional message, passed as the [`-m`] option argument

`$ git stash -u` or `$ git stash push -u`
untracked files and tracked files with uncommitted changes are stashed

`$ git stash -a` or `$ git stash push -a`
like commands with the [`-u`] option, additionally will stash the ignored files

`$ git stash pop`
a single stash state will be removed from the stash list and applied to the working directory (the working directory must match the index)

`$ git stash apply`
like the `pop` command, but the state is not removed from the stash list

`$ git stash list`
the stash entries are listed

`$ git stash clear`
all stash entries are removed

> **i** Check the Git documentation for other `stash` command options.

P3. I want to exclude some files from the version control

Solution:

Create a `.gitignore` text file in the repository directory and specify intentionally untracked files that Git should ignore. Each line in this file specifies a pattern.

Selected rules of processing the `.gitignore` file:

- Lines that are empty or begin with the "#" character are omitted.
- The "/" character is used as a directory separator. If the pattern ends with the "/", it means that all the files in the directory should be ignored (e.g., `abc/`).
- The "*" character matches zero or more characters (e.g., `*.txt` results in ignoring all files with the `.txt` extension) – anything except a slash.
- To match one of the multiple specified characters, use square brackets, e.g., `[xyz]` matches `x`, `y`, or `z`. To match one of the characters in the range, use, for example, `[0-9]` (matches digits from 0 to 9).
- To match a single character (anything except a slash), use the "?".

Examples:

- `*.a` – ignore all files with the `.a` extension
- `debug/` – ignore the `debug/` directory and its subdirectories
- `/main.c` – ignore only the `main.c` file in the main directory, don't ignore, e.g., `/src/main.c`
- `/bak/*.c` – ignore all files with the `.c` extenson in the `bak/` directory, but not files in its subdirectories (e.g. `/bak/src/main.c`)
- `/bak/**/*.c` – ignore all files with the `.c` extension in the `bak/` directory and its subdirectories

With `.gitignore` only untracked files may be ignored.

P4. I want to stop Git tracking a file

Solution: Open the Git console and execute `git rm --cached [file name]`

Example:

```
$ git rm --cached deprecated.xyz
```

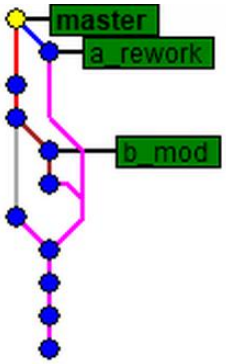
6. Further reading

- *Pro Git* Book (Chacon and Straub, 2014): git-scm.com/book/en/v2/
- An interactive graphic representation of some Git commands and concepts: ndpsoftware.com/git-cheatsheet.html
- EGit user guide: https://wiki.eclipse.org/EGit/User_Guide

7. Self-study questions

1. You want to keep some changes for later work, but remove them from your working directory. Therefore, you:
    a. stash them
    b. commit them and switch your current branch
    c. push them
    d. revert them

2. Look at the graph to the right. The newest commits are shown at the top. Branch names are given in green labels.



    a. What happened to the **b_mod** branch?
        i. It was merged into the **master** branch.
        ii. It was deleted.
        iii. It was committed to the **master** branch.
        iv. It was stashed.
    b. How many commits are there to the **master** branch?
        i. 6
        ii. 8
        iii. 9
    c. Which of the following statements is true?
        i. The **master** branch is the **a_rework's** parent branch.
        ii. The **b_mod** is the **a_rework's** feature branch.
        iii. The **master** branch has three feature branches.

3. You fetched data from the remote repository. There were some changes introduced in the feature branch on which you are working. What will you do first?
    a. try to merge fetched changes locally;
    b. commit your changes to the remote repo;
    c. discard your changes, checkout the newest commit, and start writing your code again with an up-to-date working directory.

End of Document