Lund University
Dept. of Computer Science
Mattias Nordahl

Software development in teams
EDAF45
2022–10–31

Lab 0 – Git basics

1 Introduction/Overview

We expect most students to already be somewhat familiar with Git and to have some experience in using it. We will, nevertheless, cover the basics in this optional lab. Even for those of you who are well versed in working with git, this lab can serve as a quick refresher. However, feel free to skim through, or skip altogether, topics that you already have a good understanding of.

1.1 How you will use Git in this course

We use the CS department's GitLab instance at `https://coursegit.cs.lth.se/`, where you will have your own ***lab repository*** (for working on labs) and an ***achievement repository*** (for reflection and grading). Lab materials will also be made available here. In the second half of the course, you will also get repositories for your projects here.

Important! *You have to log in at* `https://coursegit.cs.lth.se/` *before we can create and give you access to your repositories. You log in with your normal student id. Do this as soon as possible!*

In later labs you will work in pairs. For this lab, however, since it is optional, you may choose to do it on your own or with your lab partner. Most of the basics can be done and learned by working locally, but there is also a couple of repositories accessible to all students, where you can play around and do what ever you want. You do, however, have to set up SSH keys to be able to communicate with the GitLab server. See next section.

1.2 Setting up SSH keys

SSH keys are a secure method of authentication that doesn't involve your username or password, and is typically required by Git repository services such as GitLab. GitLab have their own guide on how to do this[1], but here is a short summary.

Note: *Your SSH key identifies your computer, and can be used with multiple services. If you already have one, you can re-use it for our GitLab server. However, if you are the type of person who is concerned with security, you might prefer to have separate SSH keys for each website that you want to access, e.g. one for our GitLab server, one for the official GitLab, one for GitHub and so on. For short advice on how to achieve that, check out the next section.*

To generate a new SSH key:

1. Open a terminal
2. Write the following command: `ssh-keygen -t ed25519 -C "your_email@example.com"`
3. You are asked to name the file. You can use the default, by just pressing Enter.
4. Next, you are asked for a password. You may leave it blank and just press Enter.

See example output below:

[1] `https://docs.gitlab.com/ee/user/ssh.html#generate-an-ssh-key-pair`

```
user@ubuntu:~# ssh-keygen -t ed25519 -C "your_email@example.com"
ssh-keygen -t ed25519 -Ckeygen
Generating public/private ed25519 key pair.
Enter file in which to save the key (~/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_ed25519
Your public key has been saved in ~/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:emJQ4KEgqO+zTNdwl/BL1raW3iRY1B1dpBcyaUWHm44 keygen
The key's randomart image is:
+--[ED25519 256]--+
|+   o        o=B=|
|o. o o     . ++o+|
|. . . o   . o oo.|
|.    . o +    o. |
| .  o . S +  o   |
|  .  = = = oE .  |
| .. . = + = .    |
| oo. . o o +     |
|  oo      . .    |
+----[SHA256]-----+
```

This will have generated a public and private key-pair in your `~/.ssh` folder. Next, you need to copy the contents of the ***public key*** (ends with `.pub`, contains a single line of text) to your GitLab account. For example, print the content with `cat`, then select and copy the text.

```
user@ubuntu:~# ls ~/.ssh
id_ed25519  id_ed25519.pub
user@ubuntu:~# cat ~/.ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIHtVeX4rKT51RK98rIavekxPnUumiRTzu1Z3WtKZuaON
    your_email@example.com
```

Next, log into `https://coursegit.cs.lth.se/` and open ***Preferences*** from the top-right menu. Then select ***SSH Keys*** in the left-side menu. Now, paste your public key, and optionally set a title an expiration date, and press ***Add key***.

And that's it!

1.3 Using multiple SSH keys

Using different keys for different websites is not difficult, but requires a little more setup. First, generate new keys just as in the example in the previous sections. Some additional practical tips:

- Use the comment in each key to identify which site you're using the key for. Include, e.g., your username and the website address, e.g.: `ssh-keygen -t ed25519 -C "coursegit"`

- Rather than using the default file name, give each key-pair a name that identifies which website you are using it on. For example:

```
user@ubuntu:~# ls -a ~/.ssh
.   coursegit_id25519      github_id25519      config
..  coursegit_id25519.pub  github_id25519.pub  known_hosts
```

Next, create a file in your `~/.ssh` directory called `config`. Open it in an editor and specify for each website (host) which user (git) and which SSH key to use, like below.

```
user@ubuntu:~# nano ~/.ssh/config
(Showing editor content below line)
--------------------------------------------------------
Host coursegit.cs.lth.se
    User git
    IdentityFile ~/.ssh/coursegit_ed25519
Host gitlab.com
    User git
    IdentityFile ~/.ssh/gitlab_ed25519
```

2 Git basics

This is the start of the actual lab itself.

2.1 The Git model

Git is a distributed versioning system that helps us as developers to share code and collaborate with others. Files being managed by Git are stored in a database called a repository, which keeps track of all the current and previous versions of the files.

When working with Git locally, we distinguish between the ***repository*** and the ***workspace***. The repository houses all versions of our code, and they are all immutable (cannot be changed). To work with and change our code, we copy one select version of the code (usually the latest) to the workspace. The workspace is a regular folder with regular files, that we can change in a text editor or IDE. We can edit and save files in as many steps as we want. Once we reach a point where we are happy with our changes, we copy our code in its current state to the repository, where it becomes a new immutable version. See figure X for reference.

When working with GitLab or other cloud based Git repository services, they too use the Git software, but have a ***bare repository***, which simply means that there is no workspace attached to it. We can interact with and synchronize our local repository with the remote one in the cloud, using commands that we will cover in later sections.

2.2 Configuring Git on your computer (`config`)

If you have not done so already, you should configure Git on your computer, to use your name, e-mail address and preferred text editor (e.g. nano, emacs, vim, VS Code, ...). Optionally, you may also set the default main branch name[2].

```
user@ubuntu:~# git config --global user.name "Firstname Lastname"
user@ubuntu:~# git config --global user.email "your_mail@domain.com"
user@ubuntu:~# git config --global core.editor nano
user@ubuntu:~# git config --global init.defaultBranch main
```

See also section 2.9 for other possible configurations that you may find useful.

2.3 Start working on a project (`init`)

To start working with a Git project, we can either create a new local repository, or clone an existing one. The latter will be covered in section 2.8. For now, we will focus on the former and create a new local repository. Create a new directory and initiate a new repository in it. Then have a look inside.

[2]The convention used to be to name the main branch `master`, but has since changed to `main`. If you are using an older Git version, that still uses `master` as default, this extra config line lets you change the default to `main`.

```
user@ubuntu:~# mkdir ~/git/pvg-lab0
user@ubuntu:~# cd ~/git/pvg-lab0
user@ubuntu:~/git/pvg-lab0# git init
Initialized empty Git repository in ~/git/pvg-lab0/.git/
user@ubuntu:~/git/pvg-lab0# ls -a
.  ..  .git
```

This new directory now contains a complete self-contained Git repository, and the content in the `.git` directory is where information about the repository state is stored.

2.4 Making changes, and the staging area (`add`, `commit`, `reset` and `rm`)

Files in your workspace can be added, modified and removed as you please. When you are ready to add changes permanently to the repository, they are first copied (added) to the ***staging area***, then ***committed*** to the repository. The staging area is essentially just a copy of your workspace (at least, until you've made changes in the workspace). It represents your proposal for the next committed version. So, after having changed the workspace, copying your files to the staging area is Git's way of marking files to be included in the coming commit. This is done with the `git add` command. Let's try it out.

> ***Info:*** *Since adding to the staging area is a step separate from the commit itself, it gives the user fine-grained controls of which files should be committed or not. There are, however, shortcuts that lets you add and commit all files in a single step.*

Create a `README.md` file with some arbitrary content inside it. Next, check the state of the repository with the `git status` command. The status command will inform you that you are working on the main branch, called `main`, and that there are no commits currently, but that there are ***untracked files***. Untracked simply means that the files exists in the workspace, but not in the staging area, and thus are not being tracked by Git. *Note that* `nano` *below can be any editor, e.g. emacs, vim or VS Code; Choose your preferred editor.*

```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

Next, add the `README.md` file to the staging area and re-check the status. The workspace and staging area will now be in synch.

```
user@ubuntu:~/git/pvg-lab0# git add README.md
user@ubuntu:~/git/pvg-lab0# git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:  README.md
```

We can demonstrate that the workspace and the staging area are really two separate copies of your working tree, with individual states. Make another change to your `README.md` file, and check the status again. You will see that there are now different changes to the file in both the workspace and the staging area. See ***figure 1a*** for reference.

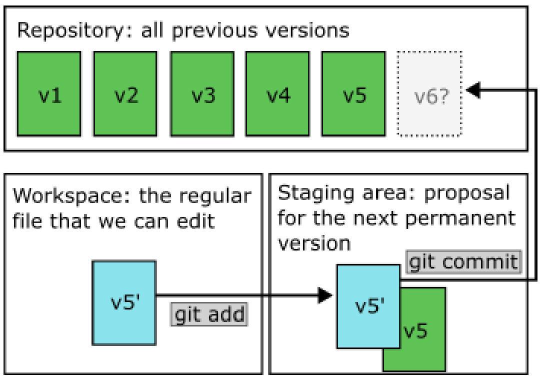
```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:  README.md

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

Add this latest edit to the staging area too, then finally, you can finalize your changes by running the `git commit` command. Git will open your configured editor where you can write your commit message. To finish, save and close the file. Alternatively, you can use the `-m` flag to give a commit message directly. Refer again to ***figure 1a***, showing how `git add` and `git commit` affect the workspace, staging area and repository.

```
user@ubuntu:~/git/pvg-lab0# git commit -m "Initial commit."
[main (root-commit) f55cf7a] Initial commit.
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

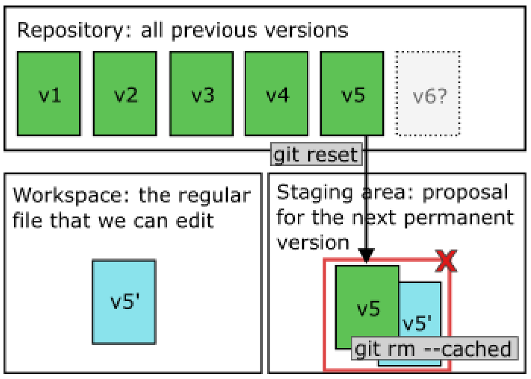
2.4.1 Removing from the staging area

Files that are `staged` (have been added to the staging area) can be `unstaged` by using either the command `git rm --cached`, `git restore --staged` or `git reset`. They do slightly different things, and can also be given arguments which might make them function more or



Repository: all previous versions
v1
v2
v3
v4
v5
v6?
Workspace: the regular file that we can edit
v5'
git add
Staging area: proposal for the next permanent version
git commit
v5'
v5

(a) `git add` copies files from the workspace to the staging area, replacing whatever version was there before. `git commit` copies files from the staging area into the repository, where it becomes a new permanent version.



Repository: all previous versions
v1
v2
v3
v4
v5
v6?
git reset
Workspace: the regular file that we can edit
v5'
Staging area: proposal for the next permanent version
v5
v5'
git rm --cached

(b) `git reset` restores files in the staging area to the version that was last fetched from the repository. `git rm --cached` removes files completely from the staging area, such that they are removed from the repository upon committing.

Figure 1: Schematic figure showing how git commands interact with and change the state of the workspace, staging area and repository.

less alike. Obviously, this can be confusing. To keep it simple, we will briefly look at only `rm --cached` and `reset`.

Both `rm --cached` and `reset` operate only on the staging area, so you won't accidentally loose your changes in the workspace[3]. The `reset` command is functionally the direct opposite of `add`; `add` copies files from your workspace to the staging area, and `reset` resets that action by copying the unmodified version from the repository back to the staging area. So, if you reset a file before a commit, that change is simply not part of the commit, but is still present in the workspace. The `rm --cached` command, on the other hand, explicitly removes files from the staging area, rather than resetting it to the unmodified repository version. This means that, if you finish your commit after removing a file with `rm --cached`, that file will no longer exist in the new repository version. It will still be present in the workspace, but be untracked. See figure 1b, which illustrates the difference.

Now, let's try it out. Modify your `README.md` file in some way, and also add a new file. The `README.md` file is already tracked (exists in the staging area), but have changes in the workspace. The new file exists only in the workspace. Try both resetting and removing the files, and ***check the status after every command*** to see the effects. You can commit if you wish.

Quiz: *Only in a certain case is `rm --cached` and `reset` equivalent. Can you see when?*

[3]Used with other parameters, both `rm` and `reset` *can* affect the workspace, so use carefully.

```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# nano SomeFile.txt
user@ubuntu:~/git/pvg-lab0# git add *
user@ubuntu:~/git/pvg-lab0# git reset README.md SomeFile.txt
user@ubuntu:~/git/pvg-lab0# git add *
user@ubuntu:~/git/pvg-lab0# git rm --cached README.md SomeFile.txt
```

2.5 Checking your work (`status`, `log`, `diff` and `blame`)

There are a couple of Git commands that lets you keep track of and analyze your work. We have already seen the `git status` command, that summarizes the current state very well, including what changes have been made in the workspace and the staging area. When we start working with a remote repository, we will see that the `status` command also displays some information about how work has progressed locally and remotely.

The `git log` command shows previous commits that have been added to the repository. It has several optional arguments for if you're looking for information about a specific commit or range of commits. Have a look at the manual by running `git log --help`.

Next, `git diff` is used to show and highlight differences between commits or entire working trees. In its base form, i.e. if called without any arguments, it compares the workspace with the staging area. You can also compare specific commits, branches (covered in the next sections) or file paths from within or outside the repository. Run `git diff --help` to see the manual.

Finally, although the name can seem a bit harsh, the `git blame` command can sometimes be useful. It shows who authored individual lines in a file. In a group project, this can be useful for finding out who has written a certain piece of code, in case you have questions or feedback to give.

2.6 Using branches (`branch` and `merge`)

One of Git's strengths is its ability to efficiently handle `branches`. Branches are like named copies of your entire work tree, including its history. The original copy is itself also a branch, conventionally called `main`. Branches enable you to break away from the main code, to a copy where you can work on your current task in isolation from other changes. Once you are done, you can `merge` your branch back into `main`. Of course, while you worked on your branch, other developers might have made changes to `main` in the meantime (covered more in later sections). If these changes are incompatible with yours, it will lead to `merge conflicts`.

2.6.1 Create a feature branch

One can deploy many `branching strategies`, depending on e.g. the size of the project or team. A common strategy is to create a new branch for every new feature. In that way, multiple people can be working separately on multiple features. You can use the `git branch` command to inspect your branches, create new ones, and switch between them. Let's try it out:

```
user@ubuntu:~/git/pvg-lab0# git branch
* main
user@ubuntu:~/git/pvg-lab0# git branch new-feature
user@ubuntu:~/git/pvg-lab0# git branch
* main
  new-feature
user@ubuntu:~/git/pvg-lab0# git checkout new-feature
Switched to branch 'new-feature'
user@ubuntu:~/git/pvg-lab0# git branch
  main
* new-feature
```

The command lists the branches that exist in your repository, and marks the current one, i.e. the branch is currently checked out in the workspace, with an asterisk (*). While you are on this new feature branch, commit a change to the README.md file, and then inspect the branches using `git branch -v` and `git show-branch`.

```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git add README.md
user@ubuntu:~/git/pvg-lab0# git commit -m 'Update README file'
[new-feature 4cda1ac] Update README file
 1 file changed, 2 insertions(+)
user@ubuntu:~/git/pvg-lab0# git branch -v
  main        a680bbb Add README file
* new-feature 4cda1ac Update README file
user@ubuntu:~/git/pvg-lab0# git show-branch
! [main] Add README file
 * [new-feature] Update README file
--
 * [new-feature] Update README file
+* [main] Add README file
```

The `-v` option means that the output will be more "verbose", i.e., give you more information. The output of the `show-branch` command consists of two parts, divided by two dashes (`--`). The first part lists existing branches, similar to `git branch`. The asterisk (*) marks the current branch and exclamation points (!) all other branches. The second part lists ***individual commits***, and indicates in the left columns which branches those commits are present on. The first commit is present on both branches, whereas the new commit is only present on the feature branch. This output becomes easier to understand if you add some more commits, so feel free to play around for a bit. You can also try to create more branches.

Now, we are happy with our new feature and want to bring it back to the main branch. We can do this using the `git merge` command. Please note, that the `merge` command merges a specified branch into the current branch; So, we first want to switch back to `main`, so that `new-feature` is merged into it, rather than the other way round[4]. After that, we'll use `git diff` to look at

[4]Of course, we might want to do the merge in the reverse order, if other developers have made changes to `main`

the difference between the branches, perform the merge, and delete the feature branch.

```
user@ubuntu:~/git/pvg-lab0# git checkout main
user@ubuntu:~/git/pvg-lab0# git diff new-feature
diff --git a/README.md b/README.md
index 7547d1d..0ee3895 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1 @@
 Some content
-
-Adding another line
user@ubuntu:~/git/pvg-lab0# git merge new-feature
Updating a680bbb..4cda1ac
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
user@ubuntu:~/git/pvg-lab0# git branch -d new-feature
Deleted branch new-feature (was 0ee3895).
```

2.6.2 Resolving conflicts

The above merge was without any conflict, meaning that Git could perform it automatically. However, when the branches being merged contain changes on the same lines, Git doesn't know what to do with them, and reports it as a ***conflict***. This can easily happen when changes are created in parallel.

You will now get to explore how conflicts can occur and how you can resolve them. Run the below commands to create a new feature branch `B1` (automatically created when the `-b` flag is given to `git checkout`) where you add some more content to the `README.md` file and then commit the change (the `-a` flag to `git commit` is short for first doing `git add`).

```
user@ubuntu:~/git/pvg-lab0# git checkout -b B1
Switched to branch 'B1'
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git commit -a -m 'Update README again'
[B1 370eab5] Update README again
 1 file changed, 2 insertions(+)
```

Then checkout `main` again and create a second feature branch `B2` where you commit a similar change on the same line in the `README.md` file. After you are done, move back to the main branch and inspect the content of the branches with `git show-branch`.

Quiz: *Why do we first switch back to* `main` *before checking out B2? Why not checkout B2 immediately?*

that we want to bring in to our feature branch.

```
user@ubuntu:~/git/pvg-lab0# git checkout main
user@ubuntu:~/git/pvg-lab0# git checkout -b B2
Switched to a new branch 'B2'
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git commit -a -m 'Update README some more'
[B2 3e9e718] Update README some more
 1 file changed, 2 insertions(+)
user@ubuntu:~/git/pvg-lab0# git checkout main
user@ubuntu:~/git/pvg-lab0# git show-branch
! [B1] Update README again
 ! [B2] Update README some more
  * [main] Update README file
---
 +  [B2] Update README some more
+   [B1] Update README again
++* [main] Update README file
```

The two feature branches now have one commit each to the same line in the same file. Run the commands below to merge `B1` into `main`, and then try to do the same with `B2`:

```
user@ubuntu:~/git/pvg-lab0# git merge B1
Updating 4cda1ac..370eab5
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
user@ubuntu:~/git/pvg-lab0# git merge B2
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
user@ubuntu:~/git/pvg-lab0# git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both modified:  README.md

no changes added to commit (use "git add" and/or "git commit -a")
user@ubuntu:~/git/pvg-lab0# cat README.md
Some content

Adding another line

<<<<<<< HEAD
Add yet another line
=======
One more line
>>>>>>> B2
```

As you can see, both changes have been put into the file, and the conflicting lines has been wrapped between `<<<<<<<< HEAD` and `>>>>>>>> B2` and divided into two parts by `=======`. The first part shows the conflicting changes from the ***current branch***, which in this case is `main`, and the second part shows the conflicting changes of the branch that the current branch ***is being merged with***, in this case `B2`. To resolve the conflict, you manually replace the whole section from the start tag to the end tag, with the content that you prefer. When you have resolved the conflict in the file, use the `git add` command again to tell Git that you have done so, and then commit your change to finish the merge:

```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git add README.md
user@ubuntu:~/git/pvg-lab0# git commit
[main 044a148] Merge branch 'B2'
```

When running just `git commit` above without the `-m` flag to give a message, you will be prompted for a message in your editor; And since you are in the middle of resolving a merge conflict a message is suggested to indicate this (i.e., `Merge branch 'B2'`). An alternative to resolving the conflict via the editor, is to resolve it using the `git mergetool` command. This command will open a tool to help you with the merge, possibly by showing you versions of the file side by side, allowing you to pull in the version you want and to edit it. You may need to configure which tool the command should use, look at the documentation for assistance (`git help mergetool`). Now let's clean things up:

```
user@ubuntu:~/git/pvg-lab0# git branch -d B1 B2
Deleted branch B1 (was 370eab5).
Deleted branch B2 (was 3e9e718).
```

2.7 Using the stash (`stash`)

Sometimes you may need to move between branches before you are ready to commit your changes on the current branch. In such cases, you can record uncommitted changes and stash them away to restore them later. Let's look at a scenario that demonstrates the usefulness of this. Run the below commands to create a branch, edit the `README.md` file, and then move back to `main`:

```
user@ubuntu:~/git/pvg-lab0# git checkout -b try-stash
Switched to a new branch 'try-stash'
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git status
On branch try-stash
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:  README.md

no changes added to commit (use "git add" and/or "git commit -a")
user@ubuntu:~/git/pvg-lab0# git checkout main
M       README.md
Switched to branch 'main'
user@ubuntu:~/git/pvg-lab0# git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:  README.md

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

The change on the `try-stash` branch "followed you" to the `main` branch. If you would like to avoid this, you can ***stash*** the changes before you switch to another branch. Switch back to the `try-stash` branch, stash your changes, then move back to `main` again:

```
user@ubuntu:~/git/pvg-lab0# git checkout try-stash
M       README.md
Switched to branch 'try-stash'
user@ubuntu:~/git/pvg-lab0# git stash
Saved working directory and index state WIP on try-stash: 044a148 Merge branch 'B2'
user@ubuntu:~/git/pvg-lab0# git status
On branch try-stash
nothing to commit, working tree clean
user@ubuntu:~/git/pvg-lab0# git checkout main
Switched to branch 'main'
user@ubuntu:~/git/pvg-lab0# git status
On branch main
nothing to commit, working tree clean
```

After you stashed the changes your working tree on the `try-stash` branch became clean and when you moved over to the main branch you did not bring any changes with you. When you are ready, you can restore the state on the `try-stash` branch. Run the below commands to move back to the `try-stash` branch, list available entries in the stash (this works as a stack, with the most recent entry on top), then ***apply*** the stash entry by explicitly naming it:

```
user@ubuntu:~/git/pvg-lab0# git checkout try-stash
user@ubuntu:~/git/pvg-lab0# git stash list
stash@{0}: WIP on try-stash: 044a148 Merge branch 'B2'
user@ubuntu:~/git/pvg-lab0# git stash apply stash@{0}
On branch try-stash
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:  README.md

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

Note that after applying the stash, the changes still remain on the list. To remove it you can use `git stash drop` to remove the top element or `git stash drop <name>` to remove a named stash entry. In this case, where we wanted to apply the top element, we could have also used `git stash pop` to both apply and drop the entry. For now, let's clean up and remove the branch:

```
user@ubuntu:~/git/pvg-lab0# git stash clear
user@ubuntu:~/git/pvg-lab0# git stash list
user@ubuntu:~/git/pvg-lab0# git reset --hard
HEAD is now at 044a148 Merge branch 'B2'
user@ubuntu:~/git/pvg-lab0# git checkout main
user@ubuntu:~/git/pvg-lab0# git branch -d try-stash
Deleted branch try-stash (was 044a148).
```

2.8 Working remotely and collaborating with others (`remote`, `clone`, `push`, `pull` and `fetch`)

Important! *Remember to set up SSH keys for GitLab, as explained in section 1.2.*

The goal when collaborating is that every team member can access and work on the team's shared code base. The way to do that using Git is to have a shared, ***remote*** repository that each user can ***clone*** locally. You then work locally, as we have been doing thus far, and ***push*** your commits to the remote repository. From there, other users can ***pull*** your commits and add them to their clone. Similarly, you can pull commits that others have pushed.

We can use the command `git remote` to inspect which references we have to remote repositories. Currently, we have none, so we get an empty response. So, we should set up a ***remote***, which can be done in two fundamentally different ways.

init + push If you have created a local repository, which you now wish to share, you must add a remote reference to an ***empty repository*** to which your local repository can be copied.

clone If there already exists a shared repository, you can ***clone*** it locally, which automatically sets up a remote reference to it.

For the purpose of this lab, we have prepared a couple of empty "playground repositories", where you can try things out in any manner that you want. This means that, if you are the first person to start using a repository, you can do the initial ***push*** to copy your local repository to the remote. If, on the other hand, another student has already done that, you can instead ***clone*** theirs. You'll find the repositories here:

`https://coursegit.cs.lth.se/edaf45/ht22-vt23/lab0-playground`

In the examples below, replace `<N>` to match the name of a real repository.

Sharing your local repository (to an empty repository)

Use `git remote add` to add a new remote reference, specifying its name `origin` (conventional name for the shared repository) and its location. Then use `git push` with the `-u` flag, which means that you set an ***upstream***, thereby linking your local branch to the remote one. The arguments `origin` and `main` are the name of the remote (that we just created) and the name of the remote branch (which, by convention, should match the local branch name), respectively.

```
user@ubuntu:~/git/pvg-lab0# git remote add origin
    git@coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git
user@ubuntu:~/git/pvg-lab0# git push -u origin main
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (13/13), 1.15 KiB | 590.00 KiB/s, done.
Total 13 (delta 1), reused 0 (delta 0)
To coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Cloning an existing repository

Simply use `git clone` and specify the location of the shared repository. This will create a new directory matching the name of the repository, into which the repository will be cloned. Make sure to not clone a repository into your already existing repository.

```
user@ubuntu:~/git/pvg-lab0# cd ~/git
user@ubuntu:~/git/pvg-lab0# git clone
    git@coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git
user@ubuntu:~/git/pvg-lab0# cd playground<N>
```

2.8.1 A reflection on branches and merging

Now that we will start working with a remote repository, we should note that we need to expand our branching model a bit, in the following way; Even if working on only one branch, `main`, there exists separate versions of `main` in your local repository and in the remote repository. Git handles this by also storing the state of the remote repository, and keeps track of both local and remote branches. Run `git branch -a -vv` (all branches and extra verbose) to see this. As you can see, there is both a local and a remote main branch, `main` and `remotes/origin/main`.

```
user@ubuntu:~/git/pvg-lab0# git branch -a -vv
* main                044a148 [origin/main] Merge branch 'B2'
  remotes/origin/main 044a148 Merge branch 'B2'
```

It is good to understand this model, as we introduce the ***pull***, ***push*** and ***fetch*** commands.

2.8.2 fetch

The `git fetch` command is used to download changes (commits) from the remote repository to your own. However, this will update ***your local copy of the remote branch***, i.e. `remotes/origin/main`, and synchronize it with the actual remote branch on GitLab. Then, in order to get those commits onto your own local main branch, you must do a ***merge***. Assuming that there actually are some changes to fetch, it might look like this:

```
user@ubuntu:~/git/pvg-lab0# git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 453 bytes | 90.00 KiB/s, done.
From coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>
   90e7680..768a712  main       -> origin/main
user@ubuntu:~/git/pvg-lab0# git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
user@ubuntu:~/git/pvg-lab0# git merge origin/main
Updating 90e7680..768a712
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
user@ubuntu:~/git/pvg-lab0# git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

After fetching changes, `git status` informs us that our local main branch is ***behind*** the remote `origin/main` by one commit. Then we do a merge, and status is back to being up-to-date.

2.8.3 pull

Often, many people don't actively use `git fetch`, but rather `git pull` instead. It can do a bit more "magic" behind the scene, that we will not go into here, but is essentially functionality equivalent to doing both a `fetch` and a `merge`. For a similar example as above, a `pull` might look like this:

```
user@ubuntu:~/git/pvg-lab0# git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 346 bytes | 24.00 KiB/s, done.
From coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>
   8d7591c..b38747f  main       -> origin/main
Updating 8d7591c..b38747f
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
user@ubuntu:~/git/pvg-lab0# git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

2.8.4 push

To copy commits in the other direction, i.e. from your local repository to the remote, you use the command `git push`. So, from modifying files in your workspace, there are three steps to take to share them; ***add*** them to the staging area, ***commit*** them to your local branch, and ***push*** the contents of that branch to the remote.

However, whenever two branches are to be merged, as we have seen, there might be conflicts. From where we are sitting, conflicts can't be handled if they occur on the remote device. So, if your repository is not in synch with the remote before pushing, Git will simply disallow it. Instead, you must first pull the latest changes from the remote, and merge them locally. Then, you can add and commit the merge, and then push. It can look like this:

```
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git commit -a -m "Updated README"
[main 1e9befb] Updated README
 1 file changed, 3 insertions(+)
user@ubuntu:~/git/pvg-lab0# git push
To coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to
    'coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

```
user@ubuntu:~/git/pvg-lab0# git pull
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
user@ubuntu:~/git/pvg-lab0# nano README.md
user@ubuntu:~/git/pvg-lab0# git add README.md
user@ubuntu:~/git/pvg-lab0# git commit -a -m "Updated README again"
[main 869fdf8] Updated README again
user@ubuntu:~/git/pvg-lab0# git push
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 779 bytes | 779.00 KiB/s, done.
Total 8 (delta 2), reused 0 (delta 0), pack-reused 0
To coursegit.cs.lth.se:edaf45/ht22-vt23/lab0-playground/playground<N>.git
   19f0e26..869fdf8  main -> main
```

2.9 Other, sometimes useful, utilities

- Feel free to keep a cheat sheet at the ready, e.g.: `https://about.gitlab.com/images/press/git-cheat-sheet.pdf`
- Some other git settings you might want to configure. Note that this is the opinions of one person, read more here and make up your own opinion:
  `https://spin.atomicobject.com/2020/05/05/git-configurations-default/`

  ```
  user@ubuntu:~# git config --global pull.rebase true
  user@ubuntu:~# git config --global fetch.prune true
  user@ubuntu:~# git config --global diff.colorMoved zebra
  ```
- ...