# Using Git on SageMath Cloud

Nico Van Cleemput

June 1, 2017

Git is a distributed versioning system. This means that each repository contains all information, and there is no real central repository. Usually there is a repository that acts as a central repository (in our case: the repository on GitHub), but all other repository contain the same information as that repository.

Git uses some terminology:

- **Git repository**: a Git repository is a directory containing a `.git` directory. All files and directories inside this folder also belong to the Git repository. A Git repository can contain files and directories which are not tracked by the versioning system.
- **Working copy**: This refers to the current state of the files in the Git repository. The changes to your working copy are not automatically stored in the history.
- **Staging area**: This contains a version of the files that will be committed to the project history with the next commit.
- **Remote repositories**: You can list any number of remote repositories. In our case there will only be the `origin` which will be on GitHub.

***Warning**: This manual is not intended to be used in shared SMC projects as SMC does not have a way to distinguish between the users.*

## 1 Preparing to use Git

If you will not create new repositories or push to existing repositories on GitHub, then you can just skip this section.

We assume that you already have a GitHub-account, so we just need to set up the local system, so Git knows who you are. Open the SMC project you will be using, start a terminal and type the following commands (using your own name and mail address).

```
git config --global user.name "Nico Van Cleemput"
git config --global user.email nico.vancleemput@gmail.com
```

## 2 Creating a new Git repository

If there is no Git repository, we will first have to create a new one. If you already have a folder containing source files that you want to act as a repository, then go to that folder. Otherwise just go to an (empty) folder where you want the repository to live. Create a new repository using the following command.

```
git init
```

The current directory has now been transformed in a Git repository. You can check that this is indeed the case by using the following command.

```
git status
```

This should list all files under the current directory as being untracked files.

If you already have any files that you want to add to the repository, you can do this using the following command replacing *files* by a list of files you want to add to the repository.

```
git add files
```

This command adds the current version of those files to the staging area. When you are satisfied with the current version in the staging area, you can commit the staging area to the project history. This is done by using the following command.

```
git commit -m "some message"
```

You can type any message you want. A typical first message is *Initial commit*.

The only thing that remains to be done at the moment is setting up the link with a repository on GitHub. First you need to create a repository on GitHub by logging in on the website and clicking the + sign in the top right corner. Select *New repository* and enter a name for your repository. You can add a description at this point, but this is not necessary. Leave all the other fields as they are and press *Create repository*.

For the examples we have used a repository with the name *repo* and the username for GitHub is *nvcleemp*.

On the next page, Github gives you the necessary commands to set up a link between your local repository and the repository on Github. First you need to set up the repository on GitHub as the origin for your local project. This is done using the following command. Note that you will need to replace *nvcleemp* by your own username on GitHub and *repo* by the name of your repository.

```
git remote add origin https://github.com/nvcleemp/repo.git
```

The second step is to synchronise the repository history of the remote repository and the local repository. The history of the remote repository is empty, so we just need to push the history from the local repository to the remote. This is done as follows:

```
git push -u origin master
```

This pushes the repository history to the origin and sets up the link between the two projects.

## 3 Using an existing repository on GitHub

If you already have a repository on GitHub, you might want to get a copy of that repository in you current project. First you need the URL of the project you will be copying. You will find this on GitHub below the right menu. For our repository the URL is `https://github.com/nvcleemp/repo.git`.

Go to the folder you want the repositor to be in. This folder does not need to be empty, but it is best if this folder does not contain any files with the same name as files which are already in the repository. Use the following command to make a new copy of the repository.

```
git clone https://github.com/nvcleemp/repo.git .
```

If you ommit the dot at the end of the command, this command will create a folder with the same name as the repository on GitHub and place everything in that folder.

## 4 Modifying files

Before you start modifying files, it is a good habit to make sure that your local repository contains the most recent version of the files. You can do this by like this:

```
git pull --all
```

For simple repositories the option `--all` is not necessary, but it will always work.

You can then use any editor you want to modify the files. Once you have made some changes which you consider to be one unit, you can put those changes in the staging area. You can do this using the following command replacing *files* by a list of files that you want to include in the next commit. This can contain both new files as well as modified files.

```
git add files
```

You can check which files will be committed with the next commit using the `status` command:

```
git status
```

This will give you an overview of which files have been changed, which files are new, which of the changes will be committed and which files are untracked.

Commiting the files is done with the following command. Using an informative commit message will make looking for errors easier.

```
git commit -m "some message"
```

If you want to type a long commit message (containing multiple lines), then you can just use the simpler version:

```
git commit
```

This will open an editor in which you can type the commit message. The convention on GitHub is that the first line is the *title* of the commit and the following lines describe what is done in the commit.

If you accidentally staged a file which should not be staged at this point, then you can undo this using the following command in which you replace *files* by a list of files that should be unstaged.

```
git reset HEAD files
```

Note that this does NOT undo the changes you made to the file. It just removes the changes from the staging area.

If at some point you do want to undo the changes you made to a file, then you can use the following command replacing *files* by a list of files that should be reverted to their previous committed state.

```
git checkout -- files
```

Once you have made a series of commits and want to store your history in the remote repository, you can push the changes by using the following command:

```
git push --all
```

For simple repositories the option `--all` is not necessary, but it will always work.

## 5 Deleting files

Sometimes a file becomes obsolete because its content was moved to different files. In this case you might want to delete this file. Note that the file is not completely lost: if it was committed at some point, it will be recorded in the versioning history, so you don't need to keep it, because you might want to see it again at some point: you can use the history for that.

To delete a file you just use the following command:

```
git rm files
```

This removes the files from you working copy and stages that removal for a commit. After removing the files you just commit the files like you would have done after an edit. If you use the `git status` command, you can see the files which are scheduled for removal in the next commit. You can combine additions of new files, edits to certain files and removals of files in one commit.

## 6 Resolving conflicts: merging

If you want to push your local repository to the remote, but the remote repository has been changed since your last pull, you will get an error message. You will first need to pull the new state and resolve any possible conflicts. If the changes do not conflict with each other, then the state of your local repository will just be updated and you can perform the push. If there is a conflict, you will see be informed so by the output of the `git pull` command.

You can view which files have conflicts using `git status`. Resolve these conflicts using an editor of your choice, stage the changes, commit the changes and push again.

## 7 Versioning history

One of the advantages of having a versioning system, is that the development process is fully documented. Especially if your commits are very atomic, it becomes easy to discover where a mistake was introduced and to revert this mistake. This section is about ways to interact with the versioning history.

## 7.1 Viewing the history

A first thing you might want to do, is to look at the versioning history. This can be done with a simple command:

```
git log
```

This shows all the commits on the current branch. The latest commits are shown at the top.

```
commit db845ff4688d6733011d9005739c445eef88cdf1
Author: Nico Van Cleemput <nico.vancleemput@gmail.com>
Date:   Thu Sep 11 14:07:52 2014 +0000

    Message 1

commit 3cb54da337a7aa82b392a7fe947d5a8f963a6975
Author: Nico Van Cleemput <nico.vancleemput@gmail.com>
Date:   Thu Sep 11 14:05:50 2014 +0000

    Message 2

commit 8a3f9b5d2f9417f05ecd49071f396ac4b7209425
Author: Craig Larson <clarson@vcu.edu>
Date:   Thu Sep 11 09:04:11 2014 +0000

    Message 3

...
```

For each commit you see the SHA1 hash (this is used to identify the different commits), the author, the date and the corresponding commit message.

## 7.2 Returning to a certain commit

Suppose we want to return to the state of the repository after commit with a given SHA1 hash: `$COMMIT`. This can simply be done using:

```
git checkout $COMMIT
```

### 7.3 Viewing the changes for a specific commit

Suppose we want to view the changes made in a specific commit. Maybe the best way to do this, is to view them on Github, since there they get visualised quite well. Looking through the versioning history occasionally is a good idea: the history can be seen as a story that explains you how a certain functionality was developed. If you want to see the changes for a specific commit with SHA1 hash `$COMMIT` on the command line, then you can use the following command:

```
git show $COMMIT
```

### 7.4 Viewing the difference between two commits

Suppose we want to see the differences in the repository after a commit with a given SHA1 hash: `$COMMIT1` and after a commit with a given SHA1 hash: `$COMMIT2`. This can simply be done using:

```
git diff $COMMIT1 $COMMIT2
```

### 7.5 Reverting a given commit

Sometimes a certain commit introduced a mistake, and you might want to revert the changes in that commit. You can do this manually, but sometimes this is too tedious, and you might want to do it automatically. This can be done with the `git revert` command. Suppose we want to revert the changes introduced in the commit with given SHA1 hash: `$COMMIT`. This can simply be done using:

```
git revert $COMMIT
```

This undoes the changes by introducing a new commit, so the old commit remains in the history and everything is well documented, so there is no risk of permanently breaking things.

## 8 Pulling with rebase

By default, if you pull your code git will fetch the commits in the remote repository and merge them with your local working copy. If both repository contained commits since the last synchronisation, then this will result in a merge commit. These merge commits can make the history difficult to read, and can make it difficult to identify the real author of a particular line of code. A much nicer way to pull the remote commits when working on the master branch is to use *rebase*. This will revert your commits, then

apply the commits of the remote repository, and finally replay your commits on top of this new working copy. The result is a much clearer history. You can achieve this using the following pull command:

```
git pull --rebase
```

You can make rebase the default pull strategy using the following command:

```
git config --global pull.rebase true
```

# 9 Handy tips

## 9.1 Getting help

You can get an overview of the Git commands using

```
git help
```

If you need help about a specific command, you can then just use

```
git help command
```

## 9.2 Setting the username for a repository

When you push to a repository, Git will ask you for a username and password. You can set the username for a repository. This way you will just need to enter the password.

First you need to know what the URL is for your remote repository. You can get this using

```
git remote -v
```

For our repository the URL is `https://github.com/nvcleemp/repo.git` and the username on GitHub is *nvcleemp*. We can now modify the URL as follows:

```
git remote set-url origin ↩
                      https://nvcleemp@github.com/nvcleemp/repo.git
```

### 9.3 Appending to a commit

If you have just made a commit and see that you forgot to include a file or did not make all changes, then you can just stage the missing files, or make the missing changes and stage the files again. Instead of just committing, you use the following command:

```
git commit --amend
```

This will combine the new changes with the last commit. if you did not provide a new commit message using the `-m` option, this will open an editor in which the previous commit message will be shown. You can then modify the commit message or just leave it as it was. If you do not stage any new changes, you can use this command to modify an incorrect commit. Note that you should not do this once you have pushed the changes to the remote. If you need to change other commits, you can use the `git rebase` command, but this is seldom a good idea, and is less trivial. In any case this should also not be done once the changes have been pushed.

### 9.4 Ignoring files

When you open a file in SMC it will create some internal hidden files with the same name, but the with `.sage-chat` and `.sage-backup` added as an extension. Since you will never commit these files, they will show up each time you perform `git status`. You can tell git to ignore these files, i.e., do not list them as untracked and do not add them to the stage unless explicitly instructed so.

Go to the root of your repository, i.e., the folder containing the `.git` folder. Create a file `.gitignore` with the following content:

```
*.sage-chat
*.sage-backup
```

Stage the file, commit it and push it to the remote, so all your repositories will automatically ignore these files.

### 9.5 Keeping a removed file

Sometimes you might want to remove a file from the repository, but keep it in your local working copy. In that case you can remove the file using:

```
git rm --cached file
```