Git & GitHub Tutorial

Software-Engineering

by

Daniel Schädler and Jean-Pierre Hotz

22. October 2018

Course: TINF17B4

Lecturer: K. Berkling, Ph.D.

Disclaimer

Since the usage of the git commandline and ui-clients for git work in basically the same way this (written) tutorial only shows you how to use Git and GitHub from the commandline. Other UIs will be shown during the presentation, though.

On the topic of commandlines: Every command / file shown in this tutorial is enclosed in a box with a black outline. In case a command is shown, any user input will be denoted by a preceding dollar sign (as it's done by Git for Windows).

1 Use your Github user in Git

To use your GitHub user in Git (and thus also link your local commits to your Github account) you can change the user name and e-mail-adress as follows:

```
$ git config --global user.name "<your Github-Username>"
$ git config --global user.email <your Github-E-Mail>
```

In case you need verification Git will ask you for valid credentials.

2 Make Git ignore certain files / folders

To make Git ignore certain files and folders you'll have to create an `.gitignore`-file. Every line in this file will represent a pattern to excluded files, also called a rule. In case you want files with a certain pattern to be only excluded in the root-directory you'll have to precede the pattern with a forward-slash `'/'`. To exclude folders you'll have to follow the pattern by a forward-slash `'/'`. To make a pattern match anything you can simply type an asterisk `'*'`. To create an exception to a general rule, you can simply give a more specific rule (for the files you want to **include**), and precede that rule with a bang `'!'`.

The `.gitignore`-file is always useful to exclude IDE-specific files (e.g. `.iml`-files in IntelliJ) or compiled binaries. An example `.gitignore`-file for a personal Java-project is shown below:

```
*.iml
target/
.idea/
!.idea/copyright/
lib/
```

3 Creation of a local repository

There are two different ways to create a local repository, which is used to track your own changes in the exisiting code.

The first option you have is to create an „empty“ repository. In case this is done in a folder, which contains files, these files are not being tracked yet. This can be done using the following command.

```
$ git init
```

The second option (which is probably used more often in teamwork), is to clone from an already existing remote repository, like an repository on GitHub or, an ordinary Git repository, you or a team member has created on a simple file server.

```
$ git clone https://github.com/<User-Name>/<Repository-Name>.git
```

4 Make versions of files

Since Git and Github is all about file versioning, we can make Git capture single versions of files with commits. Before creating commits though, we'll have to add changes that are to be included in the version we want to capture. This process is called 'staging changes' and can be done either by staging all changes, or by only staging certain files.

```
$ git add --all
$ git add <files>
```

To then create a commit with the staged changes you can use following command:

```
$ git commit -m "<Commit-message (can be multi-line)>"
```

Commit-messages are not easy to get right, especially as a beginner to Git, but good commit messages make your work with Git much easier. Especially since this allows you to review your commit history efficiently and see who changed what, and why they did so. To explain commit messages we'll have to break it down to its components. A commit message always has a caption, which is usually the first line of the actual message. This caption should be **separated** from the body of the message **by a blank line**. Also the caption should always be **capitalized**, and have a **maximum of 50 characters** in it. A caption is supposed to make sense to be filled into the sentence „If applied, this commit will _“, which means that the imperative mood is to be used.The body of the commit message should be wrapped at **72 characters**, and should explain what has been done and why this has been done. You sould omit how you achieved what you have done, since this (should be) easily readable from the changes you have made. [1]

5 Showing changes and commits

There are different ways of showing your changes and commits. To show the differences between two commits or the current changes to certain or all files you can simply use `diff`.

```
$ git diff <commit1> <commit2>
$ git diff <commit> <filesToCompare>
$ git diff
```

[1] This paragraph heavily relies on the blog posts found at `https://medium.com/@nawarpianist/git-commit-best-practices-dab8d722de99` and `https://chris.beams.io/posts/git-commit/`

The first command will list every change you have made between the given commits. The second will list every change you have made in the given files since the given commit, and the last command will list every change you have made since the last commit. Also note that there are more options on how to use this command than we show you here.

To show files which have been changed, created, deleted or moved (while the last two options only apply to tracked files) you can use the `status`-command. It also shows which files are not tracked, or which changes are currently staged to be committed.

```
$ git status
```

To show the commit history you can use the `log`-command. To look at the commit history makes sense, whenever you need to determine the hashes of certain commits (can be used e.g. for `diff`) or to see whether something has already been done by someone.

```
$ git log
```

6 Managing a remote repository

To manage a remote repository (like a repository hosted on Github) you'll have to know two basic commands. Those are push your commits (i.e. versions) to the remote repository, or pull all the commits from the repository. And those are also the names of the git-commands.

```
$ git push
$ git pull
```

In case you have created an empty repository, you'll have to add the remote repository first, though. This can be done with the `remote-add`-command. The name of a remote repository usually is `origin`.

```
$ git remote add <Name> <URL>
```

To then push you'll also have to set this repository as the repository to push to.

```
$ git push -u <Name> <Branch>
```

After this your pushes on the given branch will automatically pushed to the given remote repository.

7 Marking important points in your history

Git allows you to `tag` specific points in your history, which is usually used to mark specific release versions. Those tags can be listed with the `--list` option.

```
$ git tag --list
```

To create a tag with a specific message you can use the options `-a` and `-m`. specific release versions. In case you want to create a tag after a certain commit you can simply state this commits hash (or the beginning of it until it is unique) after the command.

```
$ git tag -a <your version> -m "<your message>" <hash of the
   commit>
```

To share tags to remote repositories (which is not done automatically) you'll either push a single tag with the following command:

```
$ git push origin <your version>
```

Or to push all tags that have not yet been pushed, you can use this command:

```
$ git push origin --tags
```

8 Branches

Git is not only useful for working alone. It is used to manage many users working in a single project on different branches. Branches are different directions the project is heading of to. These can be represented by just older or more recent code or different features that are developed at the same time. To simplify working with different half implemented features, git uses branches. A user that creates a commit on a given branch, lets call it branch A, does not affect work that is already or concurrently done on Branch B. Therefore these two branches are seperate from each other. They may come back together at some point, but they can also stay divergent all the time. Normally the use case of git is to have a master branch of the released version of the product with a develop branch that has experimental new features. Then there are different feature branches that all implement a single new feature. These branches are where the developers spent most of their time. They implement the features they want to and then merge the branch back into develop where the code waits to get released with the next version.

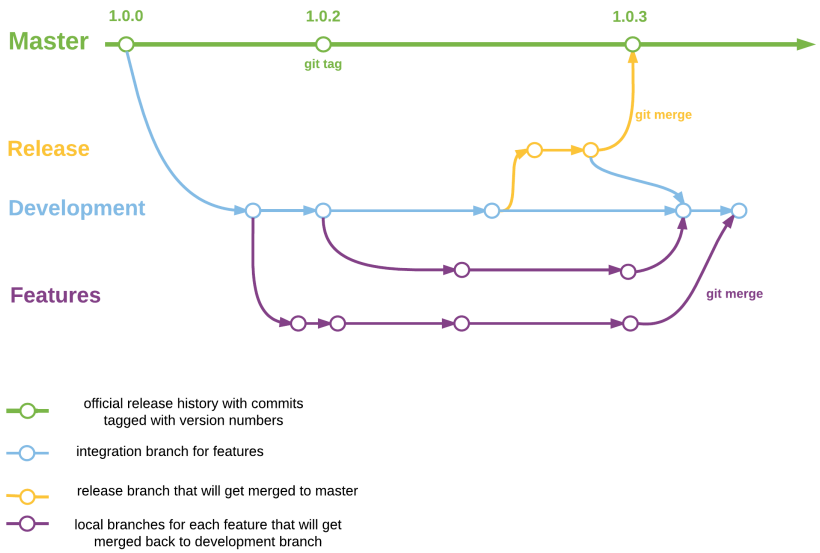

Abbildung 1: Source:
`https://help.talend.com/reader/GDDhoNyPTrERkrITzE6qPQ/QhAXF8ijSOHq~lhrVCoLKQ`

9 Merging

To bring two branches back together the user has to merge them. This can either be done by a merge commit or by rebasing the branch that needs to be merged onto the branch it needs to be merged into. A merge commit is a commit, that merges the two branches. The merge itself is started by:

```
$ git merge <branchname>
```

This merges the given branch into the current branch. When there are no conflicting changes, meaning no conflicting lines, the merge commit is empty and the branches are merged easily. When there are conflicting changes, these changes pause the merge process and need to be resolved before continuing. The conflicting lines then get marked in the File and the user has to decide, which version is right. After doing so, the merge is continues by typing

```
$ git merge --continue
```

10 Conflict resolving

If two people change the same line in a file on different branches and one tries to merge them, a conflict will happen.

```
<conflicting-file>:
To help you with git,
<<<<<<< HEAD
this line has many infos
=======
I write many different tutorials
>>>>>>> conflicting branch
```

This means that your HEAD has another version of the 2. line of file <conflicting-file> then the branch you are trying to merge. You wrote `this line has many infos` while the other branch has `I write many different tutorials` in the second line. In this case this is just a sentence that makes sense in both ways, but maybe you want to give the user both of these informations. Then you have to think about what would make sense combining the two and making it fit into the rest of the file. In this case maybe `I write many tutorials with many infos.` would combine them in a good way.

So, to resolve the conflict, you have to delete the lines with `[< / > / =]` in front of them and then decide on what the file should look like in the end. Here would be an idea of what the file could look like afterwards:

```
<conflicting-file>:
To help you with git,
I write many tutorials with many infos
```

After editing the file, you add it via

```
$ git add <conflicting-file>
```

and then go on adding more files and finally committing these changes.

11 Rebasing

When you want to avoid a merge commit, a rebase is what you want. Rebasing a branch means to change the point, where it is diverging from the other branches to the point given in the command.

```
$ git rebase <commit id / branch>
```

When typing this, git rolls back to the point where the two branches diverged, then applies all the commits that happened to the point you specified in your command and then tries to apply all the commits that happened on the current branch since it diverged.

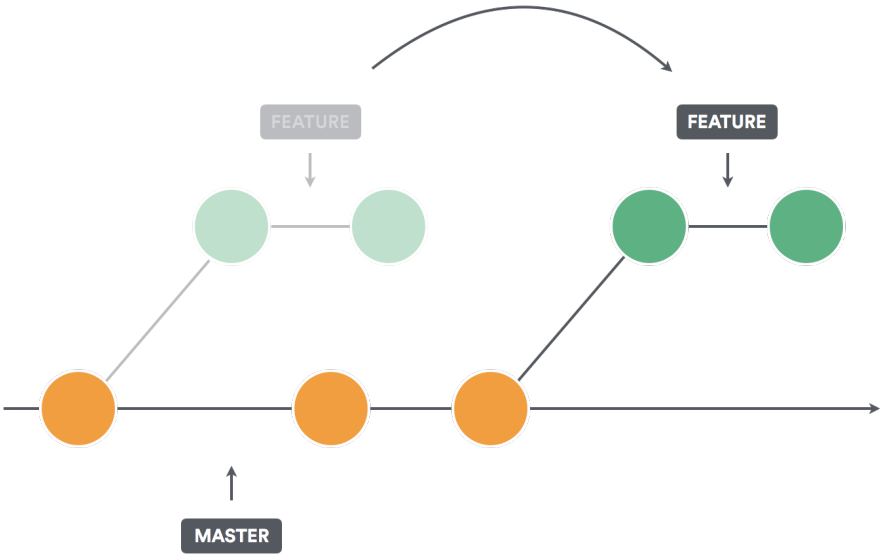

Abbildung 2: Source:
`https://www.atlassian.com/blog/sourcetree/interactive-rebase-sourcetree`

Again, if there are no conflicting changes, the rebase is successful and your branch can now be merged with the `--ff-only` option, meaning all the changes will be copied over to the branch you are merging into without creating new commits.

12 Rewriting history

Sometimes you make mistakes and you have to fix them. Therefore you can rewrite gits history. You can do that by rebasing interactively using:

```
$ git rebase -i HEAD~<number>
```

where `<number>` is the number of commits you want to go back and edit. When you rebase interactively on commandline, your editor opens a file containing something like this:

```
pick ce28e71 change gitignore intellij
pick 4067bbb Add chapter about tags
pick 78580c3 Change some slides
pick 1641359 Complete my slides
pick 091611d add current pdf version
pick db3e8b4 add Daniels part to slides

# Rebase 3a4a271..db3e8b4 onto 3a4a271 (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop to amend
# s, squash <commit> = use commit, but meld into previous commit
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a commit here THAT COMMIT WILL BE LOST.
```

On top are the commits. Below that (in this case shortened for better overview) is the commands you can do with them. To execute the commands on these commits, you simply change the word in front (normally "pick") to whatever command you want to execute. These commands are explained below in the comments of the file, therefore we will not explain them all individually.

13 GitHub (& Others)

As we are using GitHub in our group, we focus on GitHub for now. But most of the features mentioned here work pretty much the same for the other services. The first thing is

13.0.1 Branch protection

As you work with other people, it gets more and more important to secure some branches. These secured branches can (depending on the configuration) normally not be simply pushed to. Also you can just disable force pushes, making the commits immutable.

13.0.2 Pull requests

If a branch is protected from pushing, you will probably need a pull request for your changes to be added to it. A pull request, is a request you make for the other collaborators

to check your changes and review them. Once you reach the configured threshold of reviews, your changes can be merged. Within the GUI you then have the same merge options as previously explained.

13.0.3 Issues

On GitHub there is also a Ticket System for users that have problems with your project. One can raise a bug or create a ticket in general to which the collaborators can write comments and do their best to resolve the issue.