**Chatper-02 DevOps (Git and GitHub )**

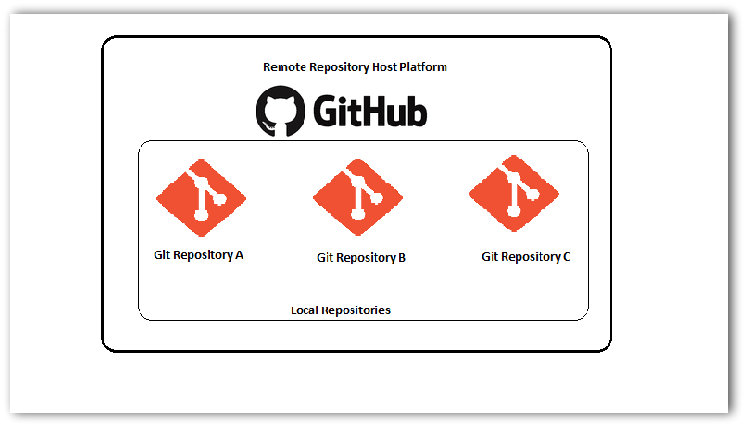
* **What is Git and GitHub?**
* **Git and GitHub life cycle**
* **Git Environment Setup**
* **Git Local Operations**
* **Git Remote Operation**

**What is Git and Github?**

In the IT industry there are two terms that are used frequently, ***Git*** and ***GitHub.***Both refer to two different entities, let us see ***what is the difference between GIT and GITHUB?***

**Git: *It is a Distributed Version Control System for tracking versions of files.***

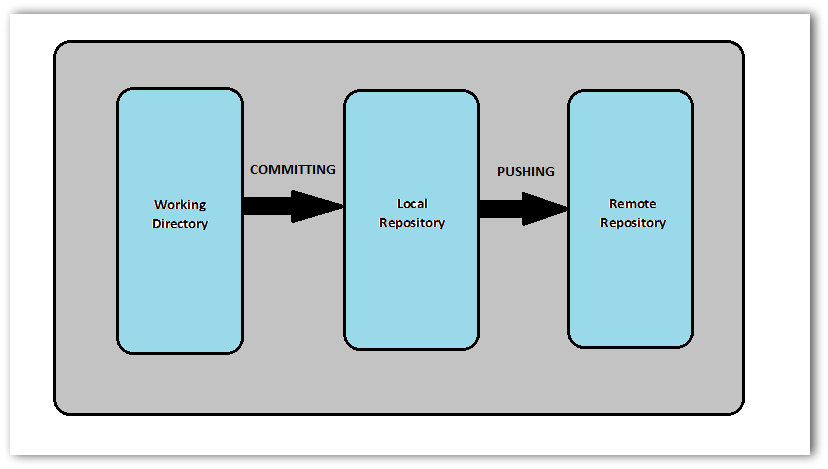
**Github: *It is  web portal and cloud hosting service for your Git repositories.***



That’s it. Git is the magic sauce that allows you to track and host versions of files on Github. ***In other words, you use commands of Git to track versions of your files. And Github is just a remote platform where these files are hosted***. However, if this explanation didn’t satisfy your queries then read on the following points elaborating more about these two terms and their usage:

***Note:****Other examples of Distributed VCS include****Bitbucket****(from the famed Atlassian family which develops Jira and Bamboo),****Beanstalk, GitLab****,****GitKraken***

***Difference between GIT and GitHub?***

1. Git allows you to track versions of your code in your local machine. However, if you want a remote backup of your code or want to publish your code to a community then you’ve to push it to Github. ***(****More about pushing code is coming in upcoming tutorials****)***
2. It’s not mandatory to use Github. Yes, you read that right. If you’re the sole person working on the project and don’t need to publish your code to the world then you can simply avoid using Github. Just track the versions of your code in local repository of your machine using Git.
3. By this point, it’d be clear that committing the code using Git in your local machine doesn’t automatically upload it to Github too. There are two distinct terms for these activities – ***Committing***and ***Pushing***. ***Committing***is capturing the changes from your working copy to your local repository. On the other hand, ***Pushing*** is uploading the captured changes from your local repository to Github**. **
4. *Neither of these processes****(Committing and Pushing)****are automatic*. You’ve to manually select the files that you want to track. In this way, you have the control of allowing and restricting the content to track and publish. Ex: Supposed you create a temporary java class to try a particular function. As this is a temporary file to write experimental code, you don’t want to commit it. So, you have the flexibility to exclude files from being tracked by committing the files manually.

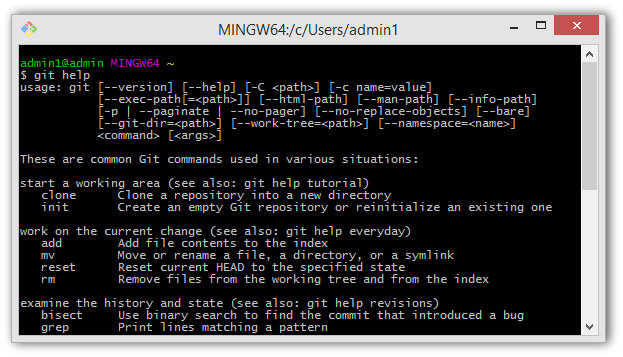
Next, let’s take a look at various ***Clients*** to use **Git**. ***Clients***here means software programs that can help you execute Git commands on a Git repository.

***Note****: Before you move forward, you will need to have Git installed on your machine. If you have not installed it yet, please follow the tutorial here:****Install Git on Windows***

***Most popular GIT Clients***

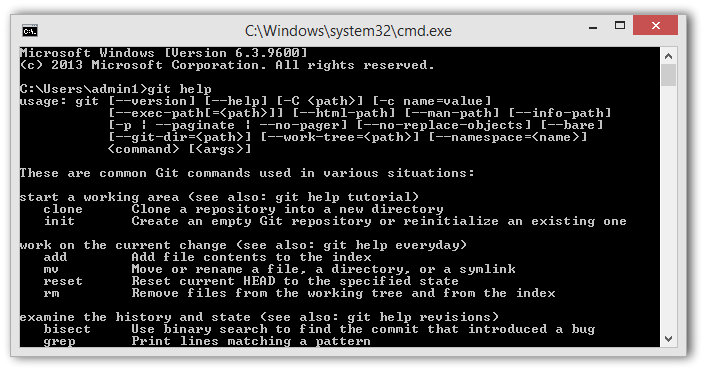
***GIT BASH***

It allows you to execute Git commands in conjunction with Unix commands in Windows environment. This tool will be downloaded automatically when you install ***GIT***. Or you can install it from the***Git Downloads*** page.



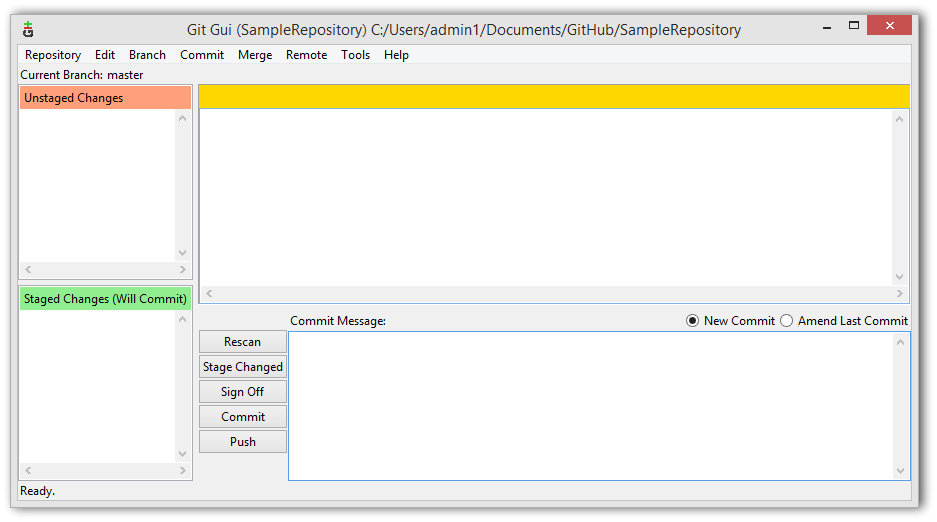
***GIT CMD***

It allows you to execute Git commands in traditional Windows Command Prompt type environment.



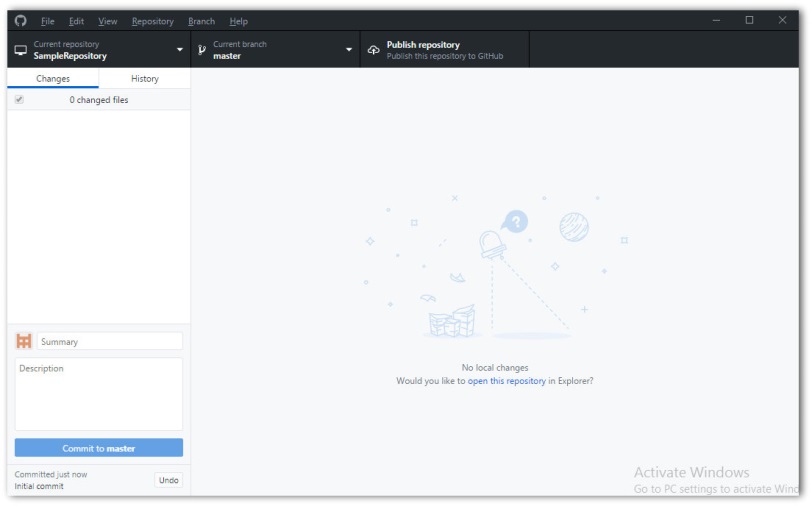
***GIT GUI***

Desktop application which lets you visualize the files under version control, modified files, difference between them and commit messages corresponding to them in addition to host of other features.



***GIT DESKTOP***

Alternate to ***Git GUI*** which lets you visualize your ***Git repository*** and its history. ***Git Desktop*** allows you to do pretty much same things as***Git GUI***. This tools is provided by GitHub and you can download it from the ***Git Desktop***page.



**Stages in GIT Life Cycle**

Files in a ***Git***project have various stages like ***Creation***, ***Modification***, ***Refactoring***, and ***Deletion***and so on. Irrespective of whether this project is tracked by Git or not, these phases are still prevalent. However when a project is under Git version control system, they are present in three major Git states in addition to these basic ones. Here are the three Git states:

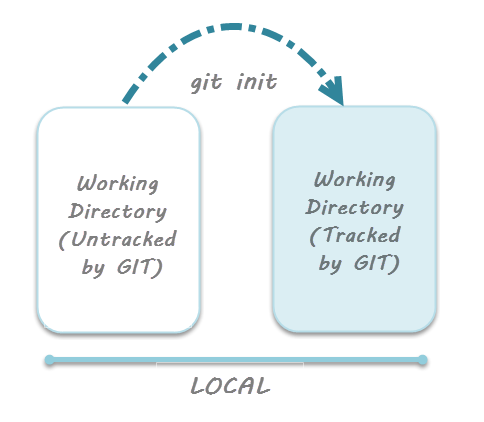
* ***Working directory***
* ***Staging area***
* ***Git directory***

These stages are the essence of Git. You get great flexibility in tracking the files due to these stages that files can reside in under Git. Let’s understand each of these states one by one.

***Working Directory***

Consider a project residing in your local system. This project may or may not be tracked by Git. In either case, this project directory is called your Working directory.

***Working directory is the directory containing hidden .git folder.***



a

***Note:******git init****– Command to initialize a Git repository*

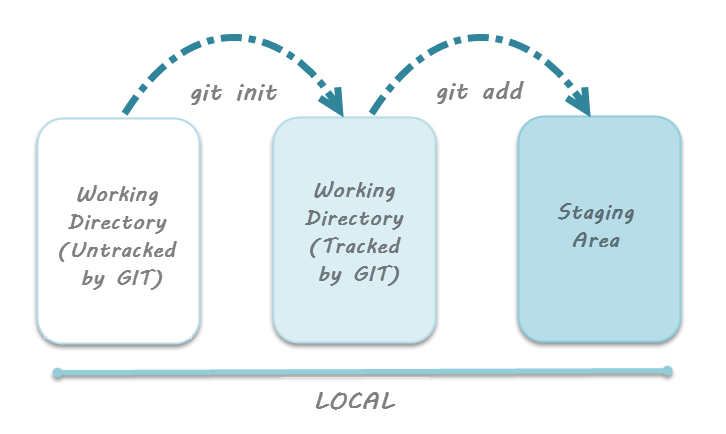
For sake of further discussion, let’s assume that this directory is now tracked by Git. That is we’ve created a Git repository in this existing project directory. So, as discussed in the tutorial on ***Creation of Git Repository***, a hidden .git folder is initialized therein. In this state, Git is just aware of the files in the project. It doesn’t track the files yet. To track the files, we’ve to commit these files by first adding the files to the staging area. This brings us to the next state in Git life-cycle.

***Staging Area***

While we’re in the working directory, we select the files that have to be tracked by Git. **Why do we need to this? Why don’t we track everything in the project?** That’s because some files in the project like***class files, log files, result files and temporary data files are dynamically generated*.** It doesn’t make sense to track the versions of these files. ***Whereas the source code files, data files, configuration files and other project artifacts contain the business logic of the application*.** These files are to be tracked by Git are thus needs to be added to the staging area.

***In other words, staging area is the playground where you group, add and organize the files to be committed to Git for tracking their versions.***

It’s important to make a quick note of the term called ***indexing***here. ***Indexing****is the process of adding files to the staging area. In other words, index constitutes of files added to the staging area***.**This term will be explained again in the coming tutorial on ***Git terminologies***.

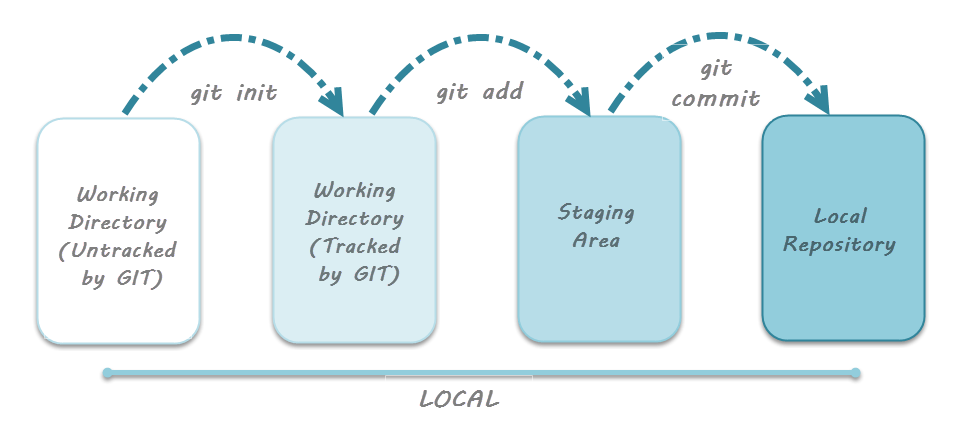


***Note:******git add****– Command to add files to staging area.*

**Git Directory**

Now that the files to be committed are grouped and ready in the staging area, we can commit these files. So, we commit this group of files along with a commit message explaining what is the commit about. Apart from commit message, this step also records the author and time of the commit. Now, a snapshot of the files in the commit is recorded by Git. The information related to this commit (*names of files committed, date and time of commit, author of commit, commit message*) is stored in the Git directory.

***Thus, Git directory is the database where metadata about project files’ history will be tracked.***

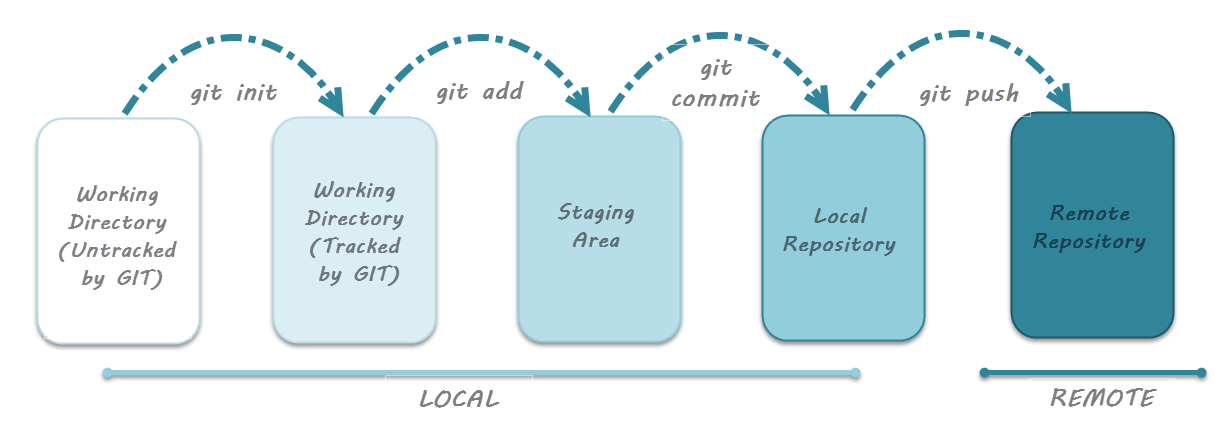


***Note:******git commit -m”your message”****– Command to commit files to Git repository with message.*

**Additional Lifecycle Stage with Github**

Now mind that we’re learning the lifecycle in Git exclusively.  That is, it’s important to note that the *three stages discussed above are only for Git and not Github*. Why? Because as explained in the tutorial on ***What is Git and Github?*** you can track versions of your files by using only Git. That is, ***Github is needed when you want to collaborate and publish your code*** to a team or community. Thus, it helps to remember that the Git cycle doesn’t conventionally involve Github.

However, we work in teams and collaborate with multiple people on a given project. This makes it imperative to understand the additional stage related to Github. While dealing with Github, there’s a concept of ***Remote repository*** and a related process called ***Pushing***the files**.**



As seen in the image above, after committing the code to the local Git repository, it has to be pushed to remote repository. *Here,****Remote repository means mirror or clone of the local Git repository in Github*.** And ***pushing means uploading the commits from local Git repository to remote repository hosted in Github*.** This will allow other collaborators to view the code. Thus, whatever changes you make in the local Git repository will be visible to other collaborators when you push your code to the remote repository. Command to push the code to remote repository in Github is ***git push***.

***Note:******git push****– Command to push commits from local Git repository to remote Git repository hosted in Github.*

***Need of Staging Area***

After learning about these stages, one might naturally ask – Why is staging area required? Why can’t we directly commit the code instead of first adding it to the staging area? Let’s understand the reason behind this with help of following points:

* ***Faster Git operations*** – Staging files regularly yields in very fast Git operations. A commit is essentially a resource-expensive interaction with the Git database. For those with background of SVN (a Client-Server model based VCS) know that every commit requires considerable time. This is because it has to first traverse the SVN commit tree to check if there have been any commits made by other users before the last commit made by us. Consequently, commit operations tend to be slow. However, in case of Git, ***staging the files doesn’t need interaction with the Git database***. It’s only when the files have to be committed that Git check for presence of commits made by other users. Thus, staging helps in recording the changes even before committing them to Git database.
* ***Visualizing the commit before actual commit*** – As discussed earlier, staging area is the state in which the files reside before they’re committed. That is, ***staging area actually lets you visualize the group of changes that will be recorded by Git***. Essentially this gives you fine-grained control over what gets committed to Git and what does not.
* ***Splitting work into separate related commits*** – Apart from the first commit when all the project files are committed at once, you should record (take snapshot of the project) at regular intervals. That is, suppose you’re working on a new feature whose timeline of development runs into several days. So, while you’re working on the feature you are required to refactor a small part of the code. In this case, you can quickly make the required changes, stage the required file and resume work on the feature. ***All of the changes i.e. feature changes and refactor code can be committed at once from the staging area*.**

**Git Local Operations**



**Terminology**

Here’s the git terminology:

* **master -** the repository’s main branch. Depending on the work flow it is the one people work on or the one where the integration happens
* **clone -** copies an existing git repository, normally from some remote location to your local environment.
* **commit -** submitting files to the repository (the local one); in other VCS it is often referred to as “checkin”
* **fetch or pull -** is like “update” or “get latest” in other VCS. The difference between fetch and pull is that pull combines both, fetching the latest code from a remote repo as well as performs the merging.
* **push -** is used to submit the code to a remote repository
* **remote -** these are “remote” locations of your repository, normally on some central server.
* **SHA -** every commit or node in the Git tree is identified by a unique SHA key. You can use them in various commands in order to manipulate a specific node.
* **head -** is a reference to the node to which our working space of the repository currently points.
* **branch -** is just like in other VCS with the difference that a branch in Git is actually nothing more special than a particular label on a given node. It is not a physical copy of the files as in other popular VCS.
* first thing you have to do is to config git with your name and email:
* $ git config --global user.name "Juri Strumpflohner"
* $ git config --global user.email "myemail@gmail.com"

## Let’s get started: Create a new Git Repository

* Before starting, lets create a new directory where the git repository will live and cd into it:
* $ mkdir mygitrepo
* $ cd mygitrepo
* Now we’re ready to initialize a brand new git repository.
* $ git init
* Initialized empty Git repository in c:/projects/mystuff/temprepos/mygitrepo/.git/
* We can check for the current status of the git repository by using
* $ git status
* *# On branch master*
* *#*
* *# Initial commit*
* *#*
* nothing to commit (create/copy files and use "git add" to track)

## Create and commit a new file

* The next step is to create a new file and add some content to it.
* $ touch hallo.txt
* $ echo Hello, world! > hallo.txt
* Again, checking for the status now reveals the following
* $ git status
* *# On branch master*
* *#*
* *# Initial commit*
* *#*
* *# Untracked files:*
* *# (use "git add <file>..." to include in what will be committed)*
* *#*
* *# hallo.txt*
* nothing added to commit but untracked files present (use "git add" to track)
* To **“register”** the file for committing we need to **add** it to git using
* $ git add hallo.txt
* Checking for the status now indicates that the file is ready to be committed:
* $ git status
* *# On branch master*
* *#*
* *# Initial commit*
* *#*
* *# Changes to be committed:*
* *# (use "git rm --cached <file>..." to unstage)*
* *#*
* *# new file: hallo.txt*
* *#*
* We can now **commit** it to the repository
* $ git commit -m "Add my first file"
* 1 file changed, 1 insertion(+)
* create mode 100644 hallo.txt

## Add another file

Lets add another file:

$ echo "Hi, I'm another file" > anotherfile.txt

$ git add .

$ git commit -m "add another file with some other content"

1 file changed, 1 insertion(+)

create mode 100644 anotherfile.txt

Btw, note that this time I used git add . which adds all files in the current directory (.).

From the point of view of the tree we now have another node and master has moved on to that one.

https://juristr.com/blog/assets/imgs/gitrepo_tree2.png

## Create a (feature)branch

Branching and merging is what makes Git so powerful and for what it has been optimized, being a distributed version control system (VCS). Indeed, **feature branches**are quite popular to be used with Git. Feature branches are created for every new kind of functionality you’re going to add to your system and they are normally deleted afterwards once the feature is merged back into the main integration branch (normally the master branch). The advantage is that you can experiment with new functionality in a separated, isolated “playground” and quickly switch back and forth to the original “master” branch when needed. Moreover, it can be easily discarded again (in case it is not needed) by simply dropping the feature branch. There’s a nice article on understanding branches in Git which you should definitely read.

But lets get started. First of all I create the new feature branch:

$ git branch my-feature-branch

Executing

$ git branch

\* master

my-feature-branch

we get a list of branches. The \* in front of master indicates that we’re currently on that branch. Lets switch to my-feature-branch instead:

$ git checkout my-feature-branch

Switched to branch 'my-feature-branch'

Again

$ git branch

master

\* my-feature-branch

*Note you can directly use the command git checkout -b my-feature-branchto*create*and*checkout*a new branch in one step.*

What’s different to other VCS is that there is only one working directory. All of your branches live in the same one and there is not a separate folder for each branch you create. Instead, when you switch between branches, Git will replace the content of your working directory to reflect the one in the branch you’re switching to.

Lets modify one of our existing files

$ echo "Hi" >> hallo.txt

$ cat hallo.txt

Hello, world!

Hi

…and then commit it to our new branch

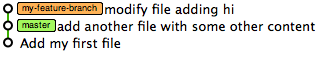
$ git commit -a -m "modify file adding hi"

2fa266a] modify file adding hi

1 file changed, 1 insertion(+)

***Note****, this time I used the git commit -a -m to add and commit a modification in one step. This works only on files that have already been added to the git repo before. New files won’t be added this way and need an explicit git add as seen before.*

What about our tree?



So far everything seems pretty normal and we still have a straight line in the tree, but note that now master remained where it was and we moved forward with my-feature-branch.

Lets switch back to master and modify the same file there as well.

$ git checkout master

Switched to branch 'master'

As expected, hallo.txt is unmodified:

$ cat hallo.txt

Hello, world!

Lets change and commit it on master as well (this will generate a nice conflict later).

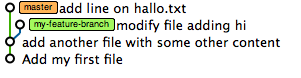
$ echo "Hi I was changed in master" >> hallo.txt

$ git commit -a -m "add line on hallo.txt"

c8616db] add line on hallo.txt

1 file changed, 1 insertion(+)

Our tree now visualizes the branch:

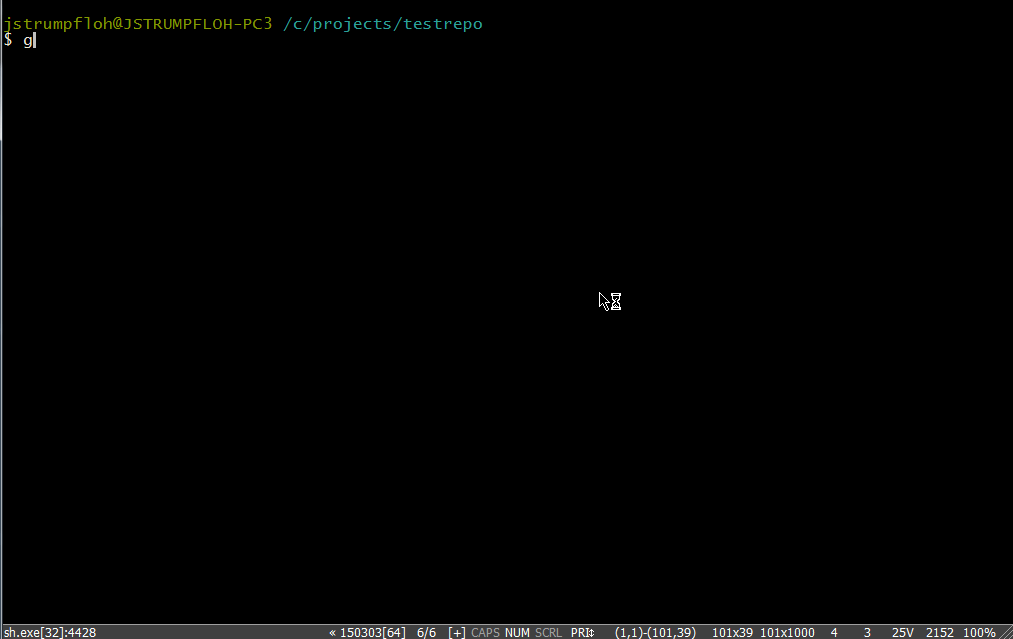


### Polishing your feature branch commits

When you create your own, personal feature branch you’re allowed to do as much commits as you want, even with kinda dirty commit messages. This is a really powerful approach as you can jump back to any point in your dev cycle. However, **once you’re ready to merge back to master** you should polish your commit history. This is done with the rebase command like this:

git rebase -i HEAD~<num-commits>

The following animated GIF shows how do do it:

Demo on cleaning up your commit history (enlarge)

## Merge and resolve conflicts

The next step would be to merge our feature branch back into master. This is done by using the merge command

$ git merge my-feature-branch

Auto-merging hallo.txt

CONFLICT (content): Merge conflict in hallo.txt

Automatic merge failed; fix conflicts and then commit the result.

As expected, we have a merge conflict in hallo.txt.

Hello, world!

<<<<<<< HEAD

Hi I was changed in master

=======

Hi

>>>>>>> my-feature-branch

Lets resolve it:

Hello, world!

Hi I was changed in master

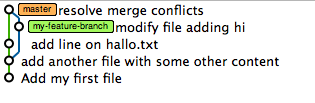
Hi

..and then commit it

$ git commit -a -m "resolve merge conflicts"

[master 6834fb2] resolve merge conflicts

The tree reflects our merge.

Fig 1: Tree state after the merge

## Jump to a certain commit

Lets assume we want to jump back to a given commit. We can use the git logcommand to get all the SHA identifiers that uniquely identify each node in the tree.

$ git log

commit 6834fb2b38d4ed12f5486ebcb6c1699fe9039e8e

Merge: c8616db 2fa266a

Author: = <juri.strumpflohner@gmail.com>

Date: Mon Apr 22 23:19:32 2013 +0200

resolve merge conflicts

commit c8616db8097e926c64bfcac4a09306839b008dc6

Author: Juri <juri.strumpflohner@gmail.com>

Date: Mon Apr 22 09:39:57 2013 +0200

add line on hallo.txt

commit 2fa266aaaa61c51bd77334516139597a727d4af1

Author: Juri <juri.strumpflohner@gmail.com>

Date: Mon Apr 22 09:24:00 2013 +0200

modify file adding hi

commit 03883808a04a268309b9b9f5c7ace651fc4f3f4b

Author: Juri <juri.strumpflohner@gmail.com>

Date: Mon Apr 22 09:13:49 2013 +0200

add another file with some other content

commit aad15dea687e46e9104db55103919d21e9be8916

Author: Juri <juri.strumpflohner@gmail.com>

Date: Mon Apr 22 08:58:51 2013 +0200

Add my first file

Take one of the identifiers (also if it isn’t the whole one, it doesn’t matter) and jump to that node by using the checkout command

$ git checkout c8616db

Note: checking out 'c8616db'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

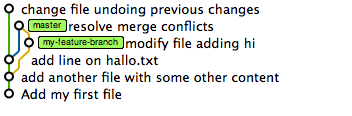
git checkout -b new\_branch\_name

HEAD is now at c8616db... add line on hallo.txt

Note the comment git prints out. What does that mean? **Detached head** means “head” is no more pointing to a branch “label” but instead to a specific commit in the tree.

*You can think of the****HEAD****as the “current branch”. When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch. […] It is possible for HEAD to refer to a specific revision that is not associated with a branch name. This situation is called a detached HEAD.*Stackoverflow Post

Basically when I now change hallo.txt and commit the change, the tree looks as follows:

Detached head state

As you can see, the newly created node has no label on it. The only reference that currently points towards it is head. However, if we now switch to master again then the previous commit will be lost as we have no way of jumping back to that tree node.

$ git checkout master

Warning: you are leaving 1 commit behind, not connected to

any of your branches:

576bcb8 change file undoing previous changes

If you want to keep them by creating a new branch, this may be a good time

to do so with:

git branch new\_branch\_name 576bcb8239e0ef49d3a6d5a227ff2d1eb73eee55

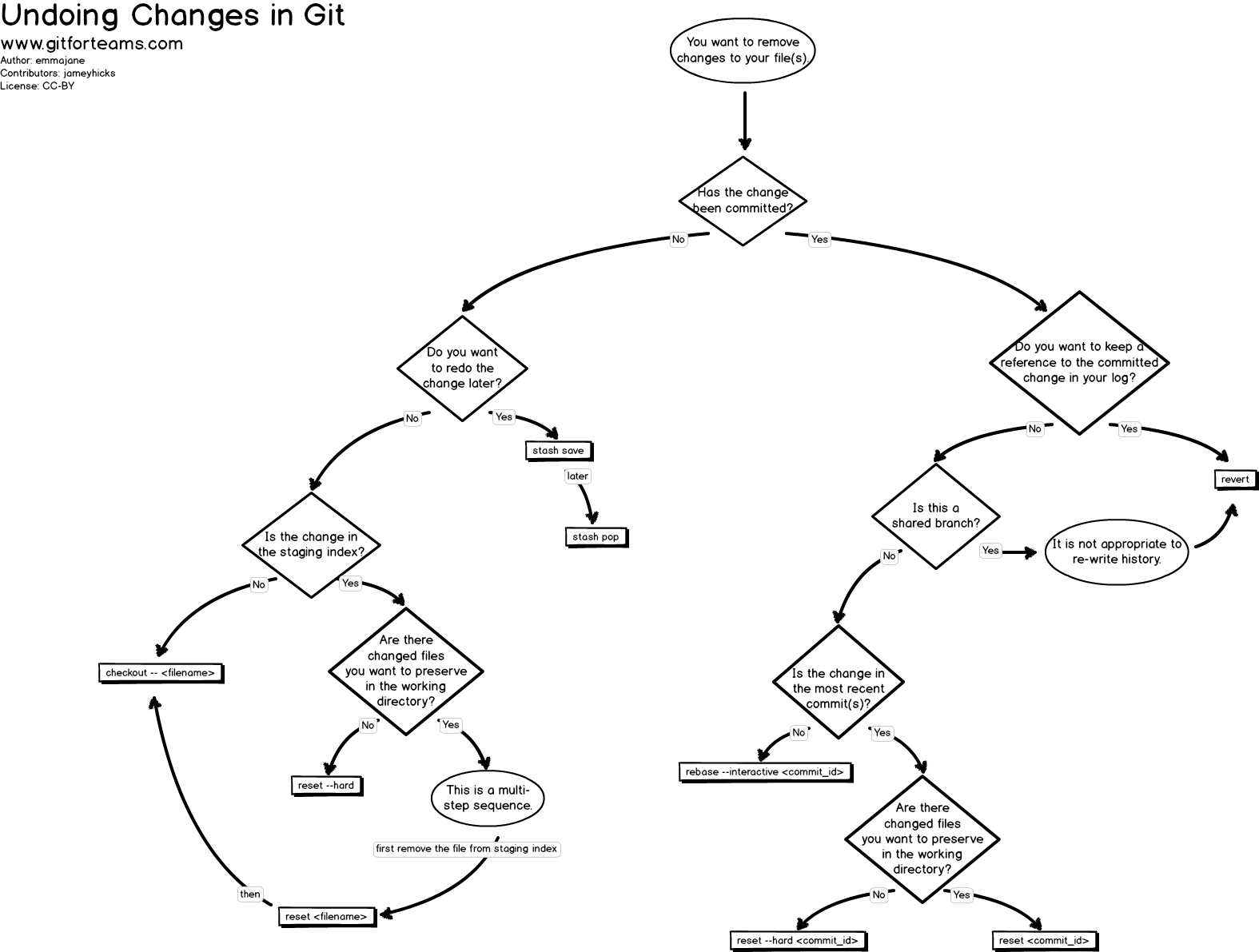
Switched to branch 'master'

And in fact, git is so kind to remind us about this fact. The tree looks now again as in figure 6.

## Undoing

Undoing is probably the operation besides committing that’s most useful about a version control system. You can quickly try out something, and if something breaks, simply dismiss your changes or jump back to a previous, working version. GitHub recently published an interesting article that might help as well: How to undo (almost) anything with Git.

Also, “emmajane” has a nice diagram illustrating the actions that need to be taken when you want to undo changes on some file(s):

By emmajane (gitforteams.com). (Click for large version)

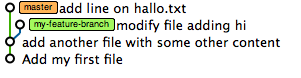
There’s more to discover on the site gitforteams.com.

### Reset hard

Jumping back is nice, but what if we want to **undo** everything back to the state before the merge of the feature branch? It is as easy as

$ git reset --hard c8616db

HEAD is now at c8616db add line on hallo.txt

The tree after the reset

The generic syntax here is git reset --hard <tag/branch/commit id>.

### Using “revert” to rollback changes the nice way

If you need to rollback an entire commit and (even worse) you might have synched it to a remote repository already, then using git reset --hard might not be so nice as in that way you’re somehow rewriting history which is a no-go if you synched your repo to a remote one already.

In such situation you can use the revert command which basically creates a new commit undoing all changes of a specific commit you specify. For instance consider you want to rollback a commit with ID 41b8684:

git revert 41b8684

### Undo Uncommitted Changes

Another common scenario of “undoing” stuff is to simply discard local, yet uncommitted changes.

#### Files Not Staged For a Commit

Assume you modified a file. Executing git status would result in

$ git status

*# On branch master*

*# 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: hallo.txt*

*#*

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

So far nothing has been added to your local Git repo, nor has it been staged (registered) for being committed. What would it mean to discard those changes?? Think about the Git tree. Simply to get (checkout) the latest version of that file, right??

Thus,

$ git checkout hallo.txt

performs our “undo”. A further

$ git status

*# On branch master*

nothing to commit, working directory clean

#### Files Staged for a Commit

The other case might be when you modified a file and already staged it for being committed through a git add commit.

$ git status

*# On branch master*

*# Changes to be committed:*

*# (use "git reset HEAD <file>..." to unstage)*

*#*

*# modified: hallo.txt*

*#*

A git checkout wouldn’t have any effect in this case, but instead (if you read what git printed on the status output) we have to do a **reset**. Why? Because the git addalready created a node in the Git tree (actually not 100% correct: see Git index vs. working tree for more details) which have not yet been committed yet, however. Therefore we need to “reset” our current pointer to HEAD which is the top of our current branch.

$ git reset HEAD hallo.txt

Unstaged changes after reset:

M hallo.txt

and consequently:

$ git status

*# On branch master*

*# 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: hallo.txt*

*#*

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

We’re now again in the state when we have local changes not yet staged for a commitand can therefore use the checkout command to discard them. A quicker way of doing so is to use the

$ git reset --hard HEAD

command which will do an un-staging + checkout in one command.

## Sharing/Synching your Repository

Ultimately we want to share our code, normally by synching it to a central repository. For doing so, we have to add a **remote**.

$ git remote add origin git@github.com:juristr/intro.js.git

To see whether I succeeded, simply type:

$ git remote -v

which lists all of the added remotes. Now we need to **publish our local branch master**to the remote repository. This is done like

$ git push -u origin master

And we’re done.

The real powerful thing is that you can add multiple different remotes. This is often used in combination with cloud hosting solutions for deploying your code on your server. For instance, you could add a remote named “deploy” which points to some cloud hosting server repository, like

$ git remote add deploy git@somecloudserver.com:juristr/myproject

and then whenever you want to publish your branch you execute a

$ git push deploy

### Cloning

Similarly it works if you’d like to start from an existing remote repository. The first step that needs to be done is to “checkout” the source code which is called **cloning** in Git terminology. So we would do something like

$ git clone git@github.com:juristr/intro.js.git

Cloning into 'intro.js'...

remote: Counting objects: 430, done.

remote: Compressing objects: 100% (293/293), done.

remote: Total 430 (delta 184), reused 363 (delta 128)

Receiving objects: 100% (430/430), 419.70 KiB | 102 KiB/s, done.

Resolving deltas: 100% (184/184), done.

This will create a folder (in this case) named “intro.js” and if we enter it

$ cd intro.js/

and check for the remotes we see that the according tracking information of the remote repository is already set up

$ git remote -v

origin git@github.com:juristr/intro.js.git (fetch)

origin git@github.com:juristr/intro.js.git (push)

We can now start the commit/branch/push cycle just normally