Git Basics

An introduction to Git VCS

Michał Kącki

Nefendi | Michał Kącki | michakacki@gmail.com

Git logo by Jason Long (source: git-scm)

Before we begin

The majority of the material covered in this presentation was based on a wonderful book written by **Scott Chacon** and **Ben Straub**. You can get it here.

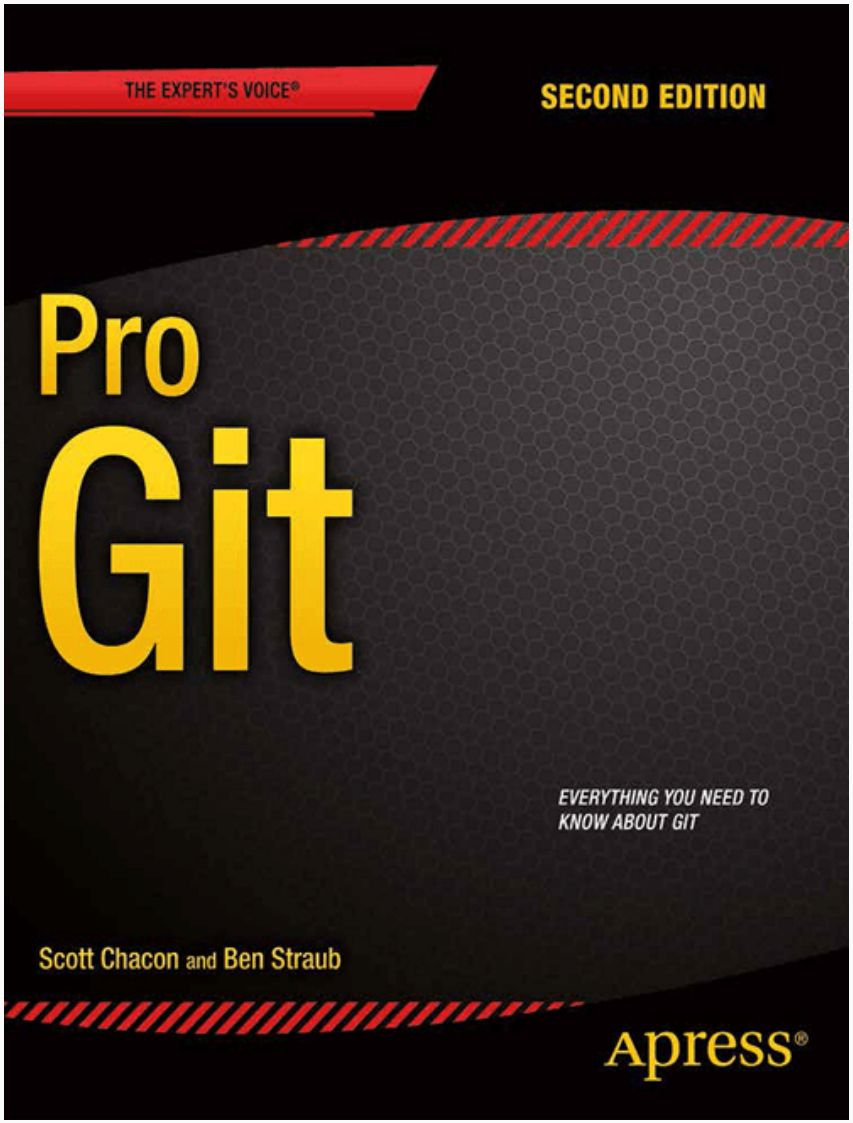

Table of contents

Introduction to VCS

Version control

Version control refers to the process of keeping track of the history of changes done to a single unit (file) or a whole collection of them. It allows saving every set of changes as a different **version** and gives the user a possibility to go back to a specific revision whenever they desire.

Version Control System

Version Control System is a stand-alone application or a program embedded in a different piece of software which realises **version control's** functionalities.

There three types of VCS, namely: local, centralised and distributed.

Local Version Control Systems

Local Version Control Systems were the first and most naive forms of VCS used long before any decent solution was created. They were realised simply by copying all the files from one place to another and optionally adding timestamps.

Later on, tools having a simplistic database were invented which improved the quality of **LVCS** just a little bit.

One program, in particular, was a very popular choice, namely **Revision Control System**. **RCS** was actually a set of UNIX commands which, under the hood, kept patch sets used for storing the history of file changes.

Pros and cons of LVCS

Advantages:

- simple to implement, all you need is an operating system

Disadvantages:

- error-prone
- no redundancy unless explicitly realised
- tedious to use

Centralised Version Control Systems

Centralised Version Control Systems improve upon LVCS by allowing more people to easily collaborate on a single project. All the versioned files are kept on a central server which can be contacted at will either to check out or check in the changes.

Most popular solutions: **Concurrent Versions System**, **Subversion**, **Perforce** and **Rational ClearCase**.

Pros and cons of CVCS

Advantages:

- allow easier collaboration
- easy to administer

Disadvantages:

- depend heavily on Internet connection
- very slow
- no redundancy unless explicitly realised*

*ClearCase uses database replication to deal with that problem.

Distributed Version Control Systems

Distributed Version Control Systems present a different approach to version control than CVCS. Instead of keeping the whole repository on one central server, it is fully mirrored by every collaborator. Such a system allows developers to work offline and communicate with an external server only to check out and check in the changes.

Most popular solutions: Git, **Mercurial**, **Bazaar** and **Darcs**.

Pros and cons of DVCS

Advantages:

- fast (connection with an external server is rarely needed)
- safe (redundancy)
- flexible (multiple workflows available)
- independent of Internet connection (most of the time)
- generates fewer conflicts (if any) at the time of merge

Disadvantages:

- not as easy to administer as CVCS
- the whole repository needs to be downloaded on each computer
- harder to learn than CVCS
- revisions are represented by long hashes rather than by incremental numbers

Git DVCS

Trivia about Git

- created in 2005 by Linus Torvalds (the author of the Linux kernel)
- the first version was written in approximately 14 days
- the maintainer as of 2005 is Junio Hamano
- the name was given by Torvalds after he wrote the first version (he described it as ‘the stupid content tracker’)

Taken from Git’s README:

‘global information tracker’: you’re in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.

‘goddamn idiotic truckload of sh*t’: when it breaks

What characterises Git?

A few non-existing switches displayed on Git's webpage:

- –fast-version-control
- –everything-is-local
- –local-branching-on-the-cheap
- –distributed-is-the-new-centralized
- –distributed-even-if-your-workflow-isnt

The main difference between Git and other VCS

While other VCS use **delta-based version control** (they keep sets of files and the changes made to them), Git realises its functionalities treating data as a series of snapshots of a miniature filesystem.

After each commit, Git looks at the state of all project files, takes a snapshot and creates a reference to it. If no changes were done to a particular file, Git simply provides a link to its previous version and does not store that file again.

How does Git know about changes?

Git uses checksums before storing anything. The checksums are generated from the contents of files by an SHA-1 hash function. Each checksum is a string of 40 hexadecimal characters, which *uniquely* identifies a given file (the probability of a collision is of the order of 10^{-45}).

Thus, even a little change made to a file results in a different hash value generated for that file, which Git immediately notices and informs the user about it.

Losing changes

It is important to say that Git rarely removes anything: it almost exclusively only adds data (removing a portion of a file is seen as a negative increment), which makes it very difficult to delete something.

What exactly is a commit?

A commit, in Git parlance, is simply a synonym for a snapshot. It holds a set of changes saved at a particular point in time. It also provides information about the author, their e-mail address and the commit's parent or parents [*sic*].

Three very important terms in Git

- Local repository (.git directory)
- Working directory (tree)
- Staging area (index)

Local repository

The heart of Git which contains the object database and all the additional information related to a project. Everything, which is stored there, is what gets uploaded to the external server and also pulled down by each collaborator.

Working directory

All the files recreated from the information stored in the compressed database (local repository), which you can modify and delete at will. To put it another way: the working tree is just a single checkout.

Staging area

The staging area (or the index) is just a file containing information about the changes which will be included in the next commit.

A (not so) quick journey through Git

Git configuration

One of the most useful commands in Git is git config. It is used to set numerous configuration option for Git **locally** (for a single project), **globally** (for the current user) or in a **system-wide** manner (for every user on a particular machine).

Configuration specificity

A more specific option always wins over a more general one. For instance, the options defined **locally** are more important than those set in a **system-wide** manner.

GIT CONFIG

Useful switches to git config

- --local (the default behaviour)
- --global
- --system
- --list — lists all settings
- --unset — unsets a variable with the given key
- --show-origin — additionally shows the file which keeps the settings
- --edit — opens a settings file in the editor of choice

Initialising a repository? What does it mean?

Git stores everything in a directory .git and to put a particular project under version control it is necessary to create that directory. Of course, it can be done manually, but Git has a command written just for that purpose: git init.

Getting started #2

git init

A useful switch to git init

git init does not have lots of switches to choose from. By far the most important one is: --bare, which creates a so-called bare repository.

An important notice about git init

Running git init inside an already existing repository will not destroy anything. It will simply refresh the list of available templates or move the contents of .git directory to the place specified via ––separate-git-dir switch (of course, if it was specified). If that is done, the only file present in the current repository will be a text file containing the path to the new location of the repository.

What about repositories that already exist?

To deal with such repositories Git has a special command: git clone. What does it do?

Firstly, it creates a directory named the same as the project being cloned (if not specified manually by the user). Then it invokes **git init** in that directory, downloads all the important data from the **remote repository** and checks out a working copy being a snapshot pointed to by the tip of the main repository **branch** (almost always **master**).

Getting started #3

GIT CLONE

Useful switches to git clone

- `--bare` — clone the repository as a bare one (no remote tracking branches)
- `--mirror` — implies `--bare` and creates a complete mirror of a repository by copying every ref in an untouched state (remote tracking branches and other refs are present, too; after running `git remote update` all refs are overwritten)
- `--origin` — change the name of the upstream repository from `origin` to the one passed to the command
- `--branch` — make local HEAD point to the specified branch instead of the one remote HEAD points to
- `--depth` — truncate the history to the specified number of commits
- `--no-tags` — do not clone any tags and set the configuration so that no tags will be pushed or pulled automatically in the future
- `--recurse-submodules` — after the clone had finished, clone and initialise all submodules

The most frequently used Git command

Git status is by far the most frequent command that you will run when managing your projects. It provides useful information about the **state of the files** in the working tree, index and diversities between your local branch and the remote one (more on that later).

The command also instructs you what you can do in many situations (e.g. upon encountering a conflict while merging two branches). As you can see, git status is an extremely useful command, which you will probably run before any other operation in git that you will want to perform.

The four states of files

- Untracked
- Unmodified (implies tracked)
- Modified
- Staged

The untracked state

Git does not know much about the file except the fact that it exists. When a file is not tracked, Git does not either save information about it in its internal database or records changes done to it.

Every new file is at first untracked and remains in that state unless explicitly made otherwise.

The unmodified state

When Git already knows about a particular file (has information about it stored in the database) and the file does not differ from its version saved in the last snapshot, that file is seen as unmodified.

The modified state

After making changes to a file which has already been saved by Git, the file enters the modified state: Git knows the previous version of the file and informs you that something has changed, and the current version differs from the one in the last snapshot.

The staged state

If a file is staged, it means that the changes introduced in a working copy will be included in the next commit. In other words, a staged file belongs to the index.

A file can enter the staged state only from either the untracked or modified state. It is quite logical that if a local version of a file does not differ from the one present in the last snapshot, that version need not be included in the next commit.

Getting started #4

GIT STATUS

Useful switches to git status

- `--short` — print the information in a short format (working copy on the right, the index on the left)
- `--branch` — print the information about branches even in short format
- `--porcelain` — the format of the printed information is adjusted to be easily parsed by various scripts or programmes
- `--verbose` — print also the changes done in the files since the last snapshot (defaults to showing the changes only for the index; specify the option two times to do this also for the working copy)

Marking the files as tracked and more

As said earlier, by default, no new file introduced to the working copy is **tracked** and such a change must be done explicitly. Performing such an operation in git can be done with the use of git add command.

But changing the state of the file from untracked to tracked is not the only thing which git add does. It is also responsible for adding the changes, which were made to already tracked files, from the working copy to the index. The command is also used for marking conflicts in files as resolved (yep, you guessed it — more on that later 🙂).

Getting started #5

GIT ADD

Useful switches to git add

- `--force` — force the adding of the ignored files
- `--interactive` — allow much more possibilities and finer control over the adding process
- `--patch` — display individual patches and ask for a permission to add them (this option is also present in the menu shown in the interactive mode)
- `--all` — add the changes of the entire working tree to the index (in older versions of Git the update was limited to the current directory and its subdirectories)
- `--no-all` — ignore adding files removed from the working copy to the index
- `--renormalize` — forcibly add already tracked files to the index while applying **clean filters** (e.g. changing line endings); do not add any untracked files

Ignoring files

Almost always there is a need to ignore certain files or types of files from being uploaded to a repository. The examples of such files can be binaries and their object files generated during a compilation, log files or documentation. For that Git has (as always) the right tool, namely the **.gitignore** file.

It is a simple text file which uses glob patterns syntax to show Git what needs to be ignored and not tracked by version control.

The structure of .gitignore

- # — comment
- * — matches any number of characters
- /name — ignores recursivity
- name/ — specifies that this is a directory not a file
- !name — negation, i.e. do not ignore that pattern (useful when one rule already ignores certain resources that we need under version control)
- [xyz] — matches one of the characters
- ? — matches one character
- [A-Z] — matches any character from the range
- dir_A/**/dir_C/ — searches for dir_C in ANY of the subdirectories of dir_A

What exactly have I changed?

Git status is a very useful command; however, sometimes we want to inspect our changes in more detail. Git diff is a tool that we can use for that. It shows what lines have been changed since the last commit and have not been staged yet (the default behaviour).

Getting started #6

git diff

Useful switches to git diff

- `--unified=<n>` — the diff has n lines of context (not changed lines present below and above of the changed ones)
- `--stat` — show the statistics of the diff
- `--word-diff` — show changed words instead of lines
- `--cached` or `--staged` — take the diff between the last commit and the index
- `--diff-filter=(A|C|D|M|R|T|U|X|B|*)` — select only those files which match the specified criteria
- `-S<string>` — show the diff only for those fragments of files which changed the number of occurences of the given string

I have created, added and diffed my changes. What's next?

When your changes are ready to become the next snapshot of your project it is time to invoke git commit. That command saves anything present in the index as a commit object so essentially a **snapshot** which you can come back to or investigate in the future.

Getting started #7

GIT COMMIT

Useful switches to git commit

- --all — automatically add to the index every unstaged change before commiting
- --reuse-message=<commit> — reuse the given commit's message, authorship and timestamp
- --author=<author> — specify the author of the commit
- --message=<message> — type a message without invoking an editor
- --no-edit — do not invoke an editor to edit a message
- --amend — create a commit in the place of the last commit (**THIS CAN POTENTIALLY REWRITE THE HISTORY**)

Removing files

To remove something from Git's version control you need to pass that information to the index. The verbose way to do that would be to remove a file or directory manually and then run git add. The creators of Git knew that it was a cumbersome solution and that is why they invented git rm. That command does those two things for you automatically. And there is no catch...

Remorse time

Okay, I lied. There is always a catch 😕. Git rm by default removes a file or directory both from its version control and from the file system. If you want to stop tracking an object but leave it on your system, you need to run git rm --cached. I apologise, now you can forgive me... Pretty please?

Getting started #8

git rm

Useful switches to git rm

- `--dry-run` — do not remove the files, just show if they exist and would be removed
- `--force` — if a file was modified or already added to the index, remove it anyway (safety catch)
- `--cached` — do not remove a file from the system

Moving files

Moving files in Git is interesting because this VCS does not track file movement. When you move or rename a file, Git just sees that there's a new file and the old one is gone. Any information about renames is generated by Git during search time not during commit time.

Enough of this superfluous knowledge. Just use git mv and be happy. Why should you use it? Simply because it is one command instead of three. Without git mv you would need to move a file manually, run git rm and then git add. If it is convenient for you, then, by all means, go for it. But do not say that you did not burn your story points because of the lack of time...

Getting started #9

git mv

Useful switches to git mv

- `--dry-run` — do nothing, just show what would be done
- `--force` — force the move even if a target with the same name already exists

...AND THE REST IS HISTORY

More than often there arises a reason to inspect the history of the project you are working on. You can dig up the plethora of valuable information with a simple yet powerful command which is git log. Its most basic version shows the full list of commits including the information about their authors, date on which they were created, their names and hashes. By default, the list is displayed in reverse chronological order as most often we are interested in the newest changes.

Am I the author or the committer?

While talking about inspecting history it is essential to mention the difference between the author and the committer of a particular commit. Fortunately, the difference is very simple — **the author** is the person who wrote the considered chunk of code and **the committer** is the one who created the commit object. Most of the time they are the same person.

In the dim and distant past, when *homo erectus* (do not laugh or I swear I will hunt you down!) tried to figure out how to make a fire, programmers used git **patches**, which were traditionally sent by e-mail. In those times, it was not rare that the author was not the committer. Happily, we are way past those times and we can eat warm food even during rainfall. Phew!

Getting started #10

GIT LOG

Useful switches to git log

- –<n> — show only n last commits
- ––oneline — print only a short version of commits' hashes and their names
- ––pretty=(oneline|short|medium|full|fuller|...) — display the history in one of the predefined ways (only a few are listed here)
- ––no–merges — omit merge commits
- ––reverse — reverse the order
- –S<string> — 'pickaxe' search, searche only for those commits which changes the number of occurrences of the specified string
- –G<regex> — 'pickaxe' search but with regex
- ––graph — print the graphical representation of the history using ASCII symbols

Unstaging files

People make mistakes. I know that is a cliché but what can you do, right? There will be times when you will be so eager to commit your changes that you will **add** everything from your working tree to the index. You will soon realise that you did not want every change in the next commit and you will need to unstage a file.

In older versions of our glorious Git doing that was rather cumbersome because you needed to run git reset HEAD <file>. What in the name of the First Sprint is that ugly construct?!

Okay, jokes aside. **Git reset** is a very powerful command which is used to manage the commit, lovely named **HEAD**, from which your working tree was initialised. But still, **git reset HEAD <file>** gives me a pain in the head. Get it?

To reset or not to reset? That is the question!

I will not teach you the power of **git reset** now because in fact there is no **git reset**. It is only yourself who resets the HEAD. Okay, I admit that was stupid but who does not like a little *Matrix* reference?

Seriously now, if you want to use **git reset** to unstage files then go for it. You will no be disappointed — the command has worked for many years. But if you have access to Git 2.23.0, then you should do better than that!

Restore yourself, I command thee!

Starting from **Git 2.23.0** we have an experimental (which means that its behaviour may change in the future) command git restore.

Normally, the command is used to manage the working tree but with the switch --staged you can manage the index. Just run git restore --staged <file>, commit, push and tell your manager that the project burns just like forests in Australia (year 2019 reference).

What about reverting modifications in the working tree?

If you are from the Mesozoic Era, there is a very high chance that your fingers will hurt from typing git checkout -- <file>. And that is good! You should like **git checkout** and it will like you in return.

Normally, the command is used to switch branches but with -- you can drop unstaged changes.

What to do if you want to be cool and modern? Use the previously mentioned **git restore**.

Warning

As you probably know, Git is going to save you in lots of situations. However, dropping the changes that have not been saved anywhere will result in losing them forever. Just be careful and you will still have your highly paid job as a software developer.

git restore

Useful switches to git restore

- `--staged` — apply the command to the index not the working copy
- `--worktree` — apply the command to the working tree (the default behaviour)
- `--source=<tree>` — specify the source tree of the restore process (by default, that is the index for the working copy and HEAD for the index)

Manipulating branches

It is about time I told you what those mysterious **branches** are. They are pointers. End of story. Go home.

Okay, okay. I will say more if you are so eager to learn. Branches are simply pointers to **commits** which **automatically** move to point to the newest commit that you have just made. That is all you need to know for now.

For manipulating branches, you may use the previously mentioned **git checkout** command. ...but we are modern, right? Let us **switch** to **Git 2.23.0** yet again in order to harness the power of the git switch. I could not resist the desire to put that pun here...

Switching — check that out!

Why did Git creators decide to introduce git switch? Probably because **git checkout** usage is twofold whereas git switch works exclusively with branches. The support for the command is still experimental but why not use it if we can?

git switch

Useful switches to git switch

- <branch> — switch to the specified branch
- --create <new-branch> — create a new branch with the specified name and switch to it
- --detach <commit-hash> — switch to the particular commit (causes the detached HEAD state)
- --force — make a switch even if the index or working copy do not match HEAD (all the changes are discarded)
- --track — specify an upstream branch which the new branch should track

More on branching

The command which primarily deals with branches is git branch. It allows you to create, list, modify and delete branches but you cannot switch to them with the use of that command. For that purpose you should use git switch or git checkout.

Even though git branch is a helpful and multipurpose command, you probably will not use it that often since other commands such as git checkout have convenient switches which use git branch internally.

git branch

Useful switches to git branch

- <no-argument> — list local branches
- <branch-name> — create a branch with the given name
- --delete <branch-name> — delete the given branch (locally)
- --move <new-branch-name> — rename a branch
- --all — list both local and remote-tracking branches
- --set-upstream-to=<upstream> — make upstream the upstream branch of the current branch

So far you have not told us about remotes only remotely...

One of the most important features of basically all Version Control Systems is the possibility to collaborate with other people. In Git, this is done by communicating with remote repositories, which each interested person should have access to.

Remote repositories are typically set up on a server but do not let the word **remote** confuse you — you can have a **remote repository** on your local machine and work with it while being in another catalogue.

Everything has its origins

If you have worked with Git before, you probably have seen the word **origin** many, many times. What is it?

Origin is the default name which Git sets to your repository after **cloning** it. Such names are just a convenient way to refer to the actual remote repositories. Those names can be changed at any time.

Have I just used a plural form? I most certainly have. Git allows you to store information about many repositories inside a single project. Why? For example, you may want to store only a small number of files from the project inside an additional repository.

How to manipulate remote repositories?

As you probably could have guessed by analysing naming conventions in Git, the command which can be used for that purpose is git remote.

This another example of a command which is not used very often. You may not use it even once during the whole lifetime of a project. The reason for that is simple — if nothing happens with a **remote repository**, there is no need to do anything with it locally.

git remote

Useful switches to git remote

- add <repo-name> <repo-url> — add a remote repository with the given name and url
- remove <repo-name> — remove the specified repository
- rename <old-repo-name> <new-repo-name> — change the name of the given repository to the new one
- show <repo-name> — print useful information about the given repository
- set–url <repo-name> <url> — set the URL of the specified repository

Biggles! Fetch... THE CUSHIONS!

If you have already cloned a repository and have a good relationship with your workmates, then you may deem them worthy and indulge yourself to download their contributions and look at them. What should you do to achieve that?

You can simply fetch them just like one of the members of the infamous Spanish Inquisition from *Monty Python* did with THE CUSHIONS.

If the Spanish Inquisition had had Git amongst their chief weapons, they probably would have used git fetch!

Getting started #15

GIT FETCH

Useful switches to git fetch

- `--all` — fetch data from all remote repositories
- `--depth=<number>` — fetch only the specified number of commits from the tip of each branch (1 means only the tip)
- `--tags` — also fetch tags in addition to anything else

Pulling vs fetching

If you sit behind a statistically important number of Git users, you will *probably* see that 90% of them pulls new changes instead of fetching them. (The number is completely made-up since I did not have the time to conduct the needed number of experiments. Also, I did not care.)

Why have I thought of such a big number? The answer is simple — computer programmers are lazy people and they automate everything they can. Remember the slide about git fetch? I typed there that the command allows you to download changes and **look** at them. While this is certainly true, you will probably wonder why you do not see **any** change in your project after running git fetch.

But why?! '*Why so serious?*,' asked Joker's father.

Fetching demystified

It turns out that git fetch downloads everything that you expects it to but it does not put the changes in your working copy. Instead, it stores them inside your **.git** directory as **remote branches**. They are the ideal copies of the true remote branches which reside inside a **remote repository**.

You can still see the contents of the branches by switching to one of them. However, to integrate them with your local ones, you need to merge the two branches.

Merging is such a fascinating process that it deserves its own presentation. A lengthy one, I might add...

So, why exactly is **git pull** so popular? You probably already know the answer — it does the merging for you!

Pulling caveats #1

Pulling may not be your good friend if you are a novice. If you do not configure your local branch correctly, you may be surprised that running a simple git pull fails.

Remember when I mentioned **upstream branches** and all that **tracking** gobbledegook? Well, it turns out I had my reasons. By default, git pull run without any argument fetches the changes, looks at the current branch that you have checked out, searches for its **upstream** branch and only then merges it with your checked out branch.

Pulling caveats #2

In other words, if your current branch does not have its **uspstream** branch specified, a simple git pull will not suffice. You will need to manually specify your remote and the name of the branch on the remote. The situation gets even worse if those two branches differ in their naming. Then you need to provide the whole **refspec** but that is a story for later.

Getting started #16

GIT PULL

Useful switches to git pull

- `--all` — fetch data from all remote repositories
- `--rebase` — rebase your local changes on top of the upstream branch (you may consider to pass **=merges** to the command to preserve more of your local history)

Additional information

There are a lot more useful switches to git pull; however, I do not find myself using them really often. A great deal of them comes from **git fetch** and **git merge** since git pull internally uses those two commands.

After you have made the desired changes locally, it is time you uploaded them to a remote repository. A Git's term for that is pushing so git push should be your tool of choice.

Pushing caveats #1

In its basic form, git push needs two arguments, namely **repository name** and **refspec**. If the **repository name** is not provided, the command tries to infer it from branch.<branch-name>.remote configuration. If that is missing, too, git push defaults to origin. When the **refspec** is missing, Git consults remote.<remote-repo-name>.push configuration. If that too is not present, the push.default configuration is used.

Starting from Git 2.0, pushing has become easier for beginners. In older versions of Git, the push.default configuration defaulted to matching value and now simple is preferred. The former causes **ALL branches with matching names on both ends** to be pushed **at once**. The latter causes the **current** branch to be pushed to its **upstream** branch **ONLY if they have the same names**.

Pushing caveats #2

This is a huge change and relaxation compared to previous behaviour. While the simple setting is a ‘safe’ (assuming you still know what you are doing) way of working with a remote repository, I prefer to change that setting to upstream. They are both similar; however, the upstream does not care if the tracking branch is named differently from the local one. In most cases, they have the same names to avoid confusion, but it depends solely on the project you are working in.

The most important thing to remember from this section is:

If you are forced to use Git < 2.0, either use the full format of git push providing a remote repository's name and refspec or change the push.default configuration.

Getting started #17

GIT PUSH

Useful switches to git push

- --all — push all branches
- --delete — delete all refs specified as arguments to the command
- --tags — push all tags in addition to all the refs specified explicitly
- --follow-tags — push all annotated tags in addition to all the refs which would be pushed without this command
- --force — push even if the remote ref is not the ancestor of the local one (do not ever use it, please!)
- --force-with-lease — the same as --force but first check if the remote's ref history matches the remote-tracking's ref one (this one you can use but still with care)
- --set-upstream — after a successful push (or nothing if a branch was up-to-date) create an upstream reference

Heute haben wir einen schönen Tag!

Often there is a need to somehow mark a specific stage of the development of the project you are working on. Such stages in IT are called **versions** and many VCS have a tool for pertaining to just that. In Git, such specific stage goes by the name tag.

Tagging allows you to freeze a specific state of the project's history and save it under a specified name. This creates a fast and easy way to go back in time to the desired version of the project and explore it.

Tags vs branches

Why should I care about some stupid tags if I have branches at my disposal?, you may ask. Well, you should care about tags since they are the right way to do it. Branches are constantly visible: you often list them using **git branch**, you see them when you want to switch to a branch in Git-repository managers such as GitHub or GitLab and so on. You probably do not want to see old refs which you need to inspect only every once in a while. This is why Git tags require a specific command to deal with them, namely **git tag**.

There is another very important reason to prefer tags to branches for versioning — tags are immutable while branches are not. And this is precisely what you want from versions — you have given your customers one version of your product and its further improvements, as well as bug fixes, should be introduced in the next version.

Two types of Git tags

Git allows you create two types of tags: lightweight and annotated.

Lightweight tags are simply pointers to particular commits. They are useful locally if you want to mark something for yourself and *generally* should not be pushed to a remote repository. This is only my advice and you will do as you wish. I think that lightweight tags are too simple objects to be shared with others and they only make remote repositories cluttered.

Another type of tag is an annotated tag. Such tags contain much more information than their smaller siblings. They have their own checksum, message, tagger's name, e-mail and date.

You can push tags like any other ref by invoking git push <remote> <tag-name> or you can use ––tags or ––follow–tags switches. The first one pushes all tags while the second one deals only with annotated ones.

git tag

Useful switches to git tag

- --list or nothing — list tags in alphabetical order, you can provide shell wildcards to filter out the results but only using the --list option
- <tag-name> — create a lightweight tag with the given name
- --annotate <tag-name> — create an annotated tag with the given name
- --message <message> — implies --annotate, attach the specified message to an annotated tag
- --force — create a tag even if a tag with the same name already exists (overwrite it)
- --delete <tag-names> — delete existing tags with the specified names

A little tip

After working some time with Git you will realise that there are things that require a considerable amount of typing and you use them quite often. This may be for example your superb format for **git log** which takes up the whole five lines in your terminal (you should probably see a doctor if that is true in your case).

Fortunately, Git gives you the possibility to create aliases with `git config --global alias.<name-of-the-alias> <command>`. One example may be — `git config --global alias.st status` which allows you to run **git status** by typing only **git st**.

I use such aliases all the time; however, if you are a beginner, you should first get acquainted with the commands before trying to create aliases for them right away.

Before we part our ways...

This is almost the end of our short (*sic!*) introduction to Git. There is only one thing left to do and that is to... TEST YOUR LUCK!

Knowledge. I meant knowledge. Test your knowledge...

Quiz time

I bet you did not expect that since I intentionally removed it from the table of contents! I am awful, I know. But what can you do?

Ladies and gentlemen, Let the 1st Git Games begin — and may the odds be ever in your favour!

Question #1

Is Git a centralised Version Control System?

a) Yes

b) No

Question #1

Is Git a centralised Version Control System?

a) Yes

b) No ✔

Question #2

How can you initialise the new repository in your **current** folder?

a) git add ––init <repo-name>
b) git init–repo
c) git init
d) git init <repo-name>

Question #2

How can you initialise the new repository in your **current** folder?

a) git add ––init <repo-name>
b) git init–repo
c) git init ✔
d) git init <repo-name>

Question #3

Can a file be modified in both the index and the working copy?

a) Yes

b) No

Question #3

Can a file be modified in both the index and the working copy?

a) Yes ✔
b) No

Question #4

You have made a mistake in your previous commit which you have not yet pushed to a remote repository. How can you fix the mistake without introducing a new commit?

a) git commit ––amend
b) git reset HEAD ––edit
c) git amend
d) git commit ––fix HEAD

Question #4

You have made a mistake in your previous commit which you have not yet pushed to a remote repository. How can you fix the mistake without introducing a new commit?

a) git commit ––amend ✔
b) git reset HEAD ––edit
c) git amend
d) git commit ––fix HEAD

Question #5

What is the **HEAD**?

a) the first commit on your current branch
b) an alias for the last remote repository that you pushed to
c) a reference to the commit from which your working tree was initialised
d) an alias to the upstream of your current branch

Question #5

What is the **HEAD**?

a) the first commit on your current branch
b) an alias for the last remote repository that you pushed to
c) a reference to the commit from which your working tree was initialised ✔
d) an alias to the upstream of your current branch

Question #6

What goes to the next commit?

a) HEAD
b) the contents of the staging area
c) the contents of the working tree
d) the contents of the .git directory

Question #6

What goes to the next commit?

a) HEAD
b) the contents of the staging area ✔
c) the contents of the working tree
d) the contents of the .git directory

Question #7

What is **origin**?

a) the first commit on your current branch
b) the tip of your current branch
c) one of the settings available for the push.default configuration
d) the default name for the first remote repository in your Git project

Question #7

What is **origin**?

a) the first commit on your current branch
b) the tip of your current branch
c) one of the settings available for the push.default configuration
d) the default name for the first remote repository in your Git project ✔

Question #8

What is the difference between git fetch and git pull?

a) there is no difference, fetch is an alias for pull
b) git pull also downloads tags in addition to everything else
c) git pull downloads and tries to merge the changes while git fetch only downloads them
d) git fetch downloads and tries to merge the changes while git pull only downloads them

Question #8

What is the difference between git fetch and git pull?

a) there is no difference, fetch is an alias for pull
b) git pull also downloads tags in addition to everything else
c) git pull downloads and tries to merge the changes while git fetch only downloads them ✔
d) git fetch downloads and tries to merge the changes while git pull only downloads them

Question #9

Can the author of a commit and its committer be two different people?

a) Yes

b) No

Question #9

Can the author of a commit and its committer be two different people?

a) Yes ✔

b) No

Question #10

Can a tracked file be omitted in your next commit?

a) Yes

b) No

Question #10

Can a tracked file be omitted in your next commit?

a) Yes ✔
b) No