|  |  |  |
| --- | --- | --- |
| Genrell | | |
| 0 | Full Manual Page: | >>>git help <verb> oder >>>git <verb> --help |
| Quick inline Help: | >>>git <verb> -h |

|  |  |  |
| --- | --- | --- |
| Start – GIT Repository | | |
| 1 | Two possible Start Scenarios: |  |
| * Declare any directory as GIT Repository | Navigate in Console to corresponding Folder:  >>>git init  Now there is a hidden .git folder created |
| * Clone a existing GIT Repository | Navigate in Console to corresponding Folder:  >>>git clone https://github.com/anyrepo [wunschname]  Now a folder anyrepo [or wunschname] is created and also a hidden .git folder with the complete clone is created |
| Note: Create a .gitignore File immediately after initializing a GIT Repository to prevent unwanted files from being included in the first commit (“initial commit”). See below. | |

|  |  |  |
| --- | --- | --- |
| .gitignore | | |
| 2 | Files that should explicitly not be tracked:  (Should not show up in >>>git status as untracked)  Templates: https://github.com/github/gitignore | Create **.gitignore** File in Working-Tree; Content e.g.:  .\* 🡪 Means ignor all files that starts with a . BUT  !.gitignore 🡪 nevertheless tracks .gitignore (force tracking)  \*.[ao] 🡪 Means ignore all files that ends wit a .a or .o  and much more patterns possible. |
| To apply a .gitignore file to all projects for a specific User: | 1. Create a .gitignore File elsewhere, for example Home directory. Name it like: .gitignore\_all\_git\_repos 2. >>>git config --global core.execluesgfile <path\_of\_gitignorefile> |

|  |  |  |
| --- | --- | --- |
| Untrack Files (Delete Only from Index) | | |
| 10 | Delete File from Staging-Area / Let GIT it not track anymore: | >>>git rm --cached datei (Now File is Untracked again) |
| For example because I forgot to add files in .gitignore, then I can delete them from the Staging-Area and add them to .gitignore.  After Untrack File:   * It is still in my working tree, they are not deleted from file system. * It is not in stageing Area / Index anymore, therefore not tracked. * And this file will no longer be Part of future Snapshots, bu still in the past snapshots. | |

|  |  |  |
| --- | --- | --- |
| Configurations | | |
| 3 | Configuration hierarchy – the specific Level will overwrite the abstract Levels:   * System (--system) – All Git-Configs applied to all Repositories on the Computer. * Global (--global) – All Git-Configs applied to certain User. * Local (--local) – All Git Configs applied to certain Repository (In this case, I must go into the Repo-Folder first).   The Configs are saved into different places/files depending on system, global or local config.  Note: it can happen that the same key/attribute is more than one time shown, because it es in 2 or 3 files set (system, global, local). Git uses the most specific value in that case. | |
| Show current Config in Comand Line:  Show current Config in Comand Line incl. Scopes:  Show current Config in Comand Line incl. Path/Place:  Show current system Config in Editor:  Show current global Config in Editor:  Show current local Config in Editor:  (Editor is set as config, see below) | >>>git config --list  >>>git config --list --show-scope  >>>git config --list --show-origin  >>>git config --system --edit  >>>git config --global --edit  >>>git config --local --edit |
| Set Username and Email:  Set Author and Email:  Set Default Editor:  Set different default Branchname (instead of Master) | >>>git config --global user.name “Yagmur Gümüs”  >>>git config --global user.email [guemuesyagmur@outlook.de](mailto:guemuesyagmur@outlook.de)  >>>git config --global author.name Yagmur  >>>git config --global author.email [guemuesyagmur@outlook.de](mailto:m.matherrs@web.de)  >>>git config --global core.editor „'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin“  >>>git config --global init.defaultBranch main |
| GIT Architecture (git status) | | |
| 5 | Ein Bild, das Pfeil enthält.  Automatisch generierte Beschreibung | |
| States from Files in Working Tree:   * Tracked (unmodified, modified, staged). * Untracked   + Either add in .gitignore so that they don’t show up in >>>git status as untracked.   + Or add into the Staging Area, to track them with >>>git add   Check State of Files: >>>git status  Short version of that: >>>git status -s | >>>git status --short |  |
| Told as a Story:   * Working-Tree and Staging-Area are synchronized in the best case (except ignored Files from .gitignore). * If I create a new File, then there is an Untracked File in my Working-Tree, because it is not in the Stating-Area. * If I modify a File, then there is a modified File in my Working-Tree, because it is not in the Staging-Area as well. * After >>>git add (or ignoring it explicitly via .gitignore), Working-Tree and Staging-Area are synchronized again, and the new File is tracked too. * After >>>git commit, Staging-Area and Repository are synchronized, respectively a Snapshot from Staging-Area is created. * Now there are three synchronized Areas: Working-Tree, Staging-Area and Repository * See below how to show differences between Working-Tree and Staging-Area as well as Staging-Area and Repository * >>>git add synchronizes **Working-Tree** with **Staging-Area** and >>>git commit synchronizes **Staging-Area** with **Repository** (as Snapshot) 🡺 Now Working-Tree, Staging-Area and Git Repository are synchronized. | |

|  |  |  |
| --- | --- | --- |
| Tracking / Staging | | |
| 6 | Add new (Untracked) or modified (Tracked) Files to Staging-Area: | >>>git add fileName | >>>git add folderName  **Everything (exclusive .gitignore):**  >>>git add . | >>>git add \* | >>>git add --all | git add -A  >>>git add -p datei.py With the -p option it is possible to select “Hunks”. This makes it possible to add parts of Files into the Staging-Area instead of the complete File. |
| If a change is made to a file after it has been Staged, it must be staged again. Otherwise, there would be three unsynchronized Areas, respectively after the commit the Working-Tree would still be modified and unsynchronized.  In other words, Git commits the staging Area not the Working Tree. |

|  |  |  |
| --- | --- | --- |
| Show Staged/Unstaged Changes | | |
| 7 | Show diffs between Working-Tree and Stagin-Area:  **Note:** If File 1 is the same in Working-Tree and Staging-Area, Result of >>>git diff is empty | **Assumption:** File 1 is staged, and a change is made in Working Tree to File 1 (File 1 is modified):  >>> git diff |
| Show diffs between Staging-Area and Repository: | **Assumption:** File 1 is committed, and a change was made to File 1 in Working Tree and was Staged into Staging-Area:  >>> git diff --staged | >>> git diff --cached |
| Show diffs between Working-Tree and Pointed Commit: | >>> git diff HEAD |
| **Notice:** This view is very confusing, I use Visual Studio Code. There are better graphical possibilities. | |

|  |  |  |
| --- | --- | --- |
| Commit | | |
| 8 | Make a Snapshot from all staged changes, respectively synchronize Staging-Area with Repository as Snapshot:  All changes that are not staged, are still in the Working-Tree and show up through >>>git status and >>>git diff because Working-Tree is not synchronized completely. | >>>git commit  🡪 The configured Editor opens and you can enter a commit message. Option -v will add die diffs into the Editor.  >>>git commit -m „Message“  🡪 Without opening a Editor |

|  |  |  |
| --- | --- | --- |
| Tagging | | |
|  | List Tag (alphabetical order)  All tags which begins with “1.8.” | >>>git tag (implicit: >>>git tag -l)  >>>git tag -l “1.8.\*” |
|  | Create lightwight Tag for last commit   * just a pointer to a specific commit | >>>git tag v1.4-lw  >>>git show v1.4-lw (shows info without extra tag info) |
|  | Create annotated Tag for last commit   * git Object (checksummed) with Meta Data | >>>git tag -a v1.4 -m “my version 1.4”  >>>git show v1.4 (show info with extra tag info) |
|  | Create Tag for previous commit – lightweight  Create Tag for previous commit - annotated | >>>git tag v1.2 <checksum\_of\_commit>  >>>git tag -a v1.2 <checksum\_of\_commit> -m “my version 2.1” |
|  | Sharing Tags (transfer tags to remote server) | >>>git push origin <tagname> (including a specific tag)  >>>git push origin --tags (including all tags) |
|  | Deleting Tags local  Deleting Tag from remote server | >>>git tag -d <tagname>  >>>git push origin --delete <tagname> |
|  | Checkout a Tag | ToDo. Corresponds to Branches (ProGit S. 60)  Udemy: Git Essentials for Beginners (3 Courses in 1 with Live Labs) Abschnitt 52 |

|  |  |  |
| --- | --- | --- |
| GitHUB | | |
|  |  | |
|  | Add remote Repository: (Configure destination of remote repository)  Show remote Repository:  Rename default Branch from master to main:   * Because remote branch default Name is also main   Push to remote first time:   * -u: add upstream (tracking) reference (local-main-branch corresponds to remote-main-branch) (git knows now, if I pull changes, I want them from remote main branche) * origin: Name (Alias) from remote Repository instead of the hole url   (I “lable” the url as origin, so that origin is now the short name)   * main: Branch from remote Repository   Push to remote:  Pull from remote: | >>>git remote add origin <link>  >>>git remote -v  >>>git branch -M main  >>>git push -u origin main  >>>git push origin main  >>>git pull |
|  | * As I understood, I connect local branches with remote branches to “track” them. * In upper default case I “connected” my local main (previously master) Branch with a newly created remote main Branch. * Now if I push and pull, I will always track only for that Branch * What is if I have another Branch locally? Do I add this to “track” to? | |

|  |  |  |
| --- | --- | --- |
| Delete Files – From Working-Tree AND Repository | | |
| 9 | Variante 1  Remove File (File will be removed from Working tree):  Stage change (File will be removed from Staging-Area):  Commit change (File is gone and no longer tracked):  Variante 2:  Remove File from Working-Tree and Staging-Area:  Commit change (File is gone and no longer tracked): | >>>rm datei.py ("Changes not staged for commit")  >>>git add datei.py ("Changes to be committed")  >>>git commit -m „Message“  >>>git rm datei.py ("Changes to be committed")  >>>git commit -m „Message“ |
| **Note:** The deleted File is still included in the previous commits (Snapshots) but will no longer be part of the future commits (Snapshots). File is deleted from Working-Tree and Staging-Area as well as no longer part of Snapshots from now on. Git Status will no longer show as Untracked File.  **Note:** If File is modified/staged after it was staged as deleted, the deletion must be forced by the Option -f  **Note:** Removing a directory with option -r | |

|  |  |  |
| --- | --- | --- |
| Move Files / Rename Files | | |
| 11 | Variante 1 - short  Rename a file in the same folder:  Variante 2 - long  Rename a file in the same folder:  (Result is same as Variante 2) | >>>git mv eins.py einseins.py (Renamed and already staged)  >>>git commit -m “msg”  >>>mv eins.py einseins.py (eins.py is deleted File (not staged) and einseins.py is Untracked File)  >>>git rm eins.py (eins.py is deleted File (staged) and einseins.py is Untracked File)  >>>git add einseins.py (eins.py is automatically detected as renamed to einseins.py)  >>>git commit -m “msg” |
| Absolutely same logic as above if I move files with/without renaming in other folders!!! See below: | |
| Variante 1 – short without rename  Variante 1 – short with rename | >>>git mv eins.py subfolder/eins.py (only moved and already staged)  >>>git commit -m “msg”  >>>git mv eins.py subfolder/einseins.py (Renamed + moved and already staged)  >>>git commit -m “msg” |

|  |  |  |
| --- | --- | --- |
| Commit History – Formatting Options | | |
| 12 | Show Commit History: | >>>git log |
| In detail option --pretty: | >>>git log --pretty=oneline  Values: oneline, short, full, fuller, format\*  \* to be specified: …=format:“%h - %an, %ar : %s“  Ein Bild, das Tisch enthält.  Automatisch generierte BeschreibungEin Bild, das Tisch enthält.  Automatisch generierte Beschreibung |
| **Comprehension:**  >>>git log shows the Commit-History (Chain) from **HEAD** **along the chain** to the root 🡪 Same as: >>>git log HEAD.   * If branch MAIN is selected, HEAD is set to newest commit of branch MAIN: >>>git log main. In this case >>>git log HEAD und >>>git log MAIN are the same!   It is possible to show the Commit-History from another branch beginning from its HEAD without changing the branch: >>>git log branch\_name  **Best Practice for me**: **>>>git log --oneline --graph --all** 🡪 From all branch “heads” to the root as a Graph | |

|  |  |  |
| --- | --- | --- |
| Commit History - Limiting Options | | |
| 13 | The last two commits: | >>>git log -2 |
| Commits of the last two weeks: | >>>git log --since=2.weeks  Flexibel anwendbar: 1.day, yesterday, 2021-03.31, …  Synnoym: --after |
| Commits until two days: | >>>git log --until=2.days  Flexibel anwendbar: yesterday, 2.weeks, …  Synonym: --before |
| Commits from a certain Author: | >>>git log --author=Yagmur  >>>git log --author=Yagmur --author=Ilhan   * Yagmur **UND/ODER** ILHAN   >>>git log --author=Yagmur --author=Ilhan --all-match   * Yagmur **UND** Ilhan (Hier leer, da nicht möglich) |
| Commits from a certain Comitter: | >>>git log --committer=Yagmur  >>>git log -- committer =Yagmur --author=Ilhan   * Yagmur **UND/ODER** ILHAN   >>>git log -- committer =Yagmur --author=Ilhan --all-match  Yagmur **UND** Ilhan (Hier leer, da nicht möglich) |
| Commits with certain Text in the Commit Messages | >>>git log --grep=Text  >>>git log --grep=Text1 --grep=Text2   * Text1 **UND/ODER** Text2   >>>git log --grep=Text1 --grep=Text2 –all-match   * Text1 **UND** Text2 |
| Above it is an OR, with this option it becomes an AND (all criteria must match to be shown in the result. | --all-match |
| Commits where a specific file was part of:  Commits from until: | >>>git log datei.py  >>>git log <SHA1>..HEAD (from given HashCode until HEAD) |
| Recherche, was ist das (GITs Pickle): | >>>git log -S function\_name |
| Changed Files/Folders (always LAST Option) | >>>git log …. --Pfad |

|  |  |  |
| --- | --- | --- |
| Discard Changes from File in Working Tree (Reset to last committed state) | | |
| 16 | Restore last State of a File: | >>>git checkout -- datei.py (OLD)  >>>git restore datei.py (NEW) |
| Use command for a file which is not in statging Area to “load” the last version as it was last committed. The file is then not modified anymore but all changes are deleted since the last commit. | |

|  |  |  |
| --- | --- | --- |
| Unstage File from Stageing Area (“Undo Staging”) | | |
| 15 | Unstage a File from Staging-Area: | >>>git reset HEAD datei.py (OLD)  >>>git restore --staged datei.py (NEW) |
| Use command for a file which is in staging Area to unstage it again, so that it is shown as “Changes not staged for commit” and in status modified. | |

|  |  |  |  |
| --- | --- | --- | --- |
| Change last Commit | | | |
| 14 | Preliminary remark:   * It is highly not recommended to make any changes to commits other than the last/newest commit! * Because all commits have a Parent/Child relationship, and the child SHA-1 is part of the new commit * Only the last commit has no Parent and is therefore not a child of another commit and therefore not dependent of any other commits. | | |
| Change last Commit Message:  (Replace last Commit with this commit!) | >>>git commit --amend -m „Other Message“ | |
| Forgot to Stage a File (too early committed):  (Replace last Commit with this commit!) | >>>git add weitere\_datei.py  >>>git commit --amend -m „same message as before“ | |
| Extend last committed File and Replace last commit:  (Replace last Commit with this commit!) | Change datei.py  >>>git add datei.py  >>>>>>git commit --amend -m „same message as before + more“ | |
| What happens: „The last Commit will be REPAIRED“ – **More Correct:** The last Commit will be REPLACED!  If i make a commit I want to change, then stage the changes and commit with --amend again!   * To prevent commits like: „Ups, File forgotten“ or „Ups, typo in Message“ | | |
| Revert last commit: | | >>>git revert HEAD |
| What happens is not that the last commit is deleted but that there is an additional new commit with the opposite content as in the last commt. | | |

|  |  |  |
| --- | --- | --- |
| Go back to a Snapshot | | |
| 17 | Assuming that only the main branch is available and there are more than 2 Snapshots/Commits in the “chain”: | |
| Go back to a Snapshot (Commit) with a certain Hash-Code:  (The HEAD is set to this Snapshot and all the following Commits are gone) | >>>git checkout 1234567\*  \*SHA-1 from the Snapshot/Commit |
| Switch back to main Branche:  (Set the HEAD again to the newest commit from branch) | >>>git switch main  (or >>>git checkout newest\_hashcode ?) |
| **Important:** If I want to go back to a Snapshot and do changes, then there are two scenarios:   1. Go back to a Snapshot and work in another direction, because “I failed my way”:    1. Go back to the particular Snapshot.    2. Create a new branch and switch to this **immediately.**    3. Delete the current MAIN branch and rename the new branch to MAIN.    4. **ATTENTION**: All changes up from the checked-out Snapshot in the new branch are LOST. Maybe, rename the “old MAIN” to “to be deleted” and delete it later, if I am sure about that. 2. Go back to a Snapshot, do some changes here and keep both, this one and the HEAD of MAIN itself:    1. Go back to the particular Snapshot.    2. Create a new branch and switch to this **immediately.**    3. Do some changes and switch back to MAIN Branch    4. Merge new branch into the MAIN branch    5. **NOTE**: Not the best approach. Why would you do that?   If i want to make changes in the checked-out Snapshot, it is important to create a new branch first, switch to it and commit the changes in this new branch. Otherwise there will be a conflict, respectively, if I switch back to the “real” HEAD of the MAIN, I can not identify the commits made before starting from the checkout Snapshot (I have to search for the SHA-1 from that commit manually by digging). | |

|  |  |  |
| --- | --- | --- |
| Branches and Merging them | | |
| 18 |  | * Create new feature Branch from the main branch * Make changes in the new branch and commit them (multiple commits possible) * Switch back to main branch (GIT Wording: main is here now receiving branch) * Merge feature branch to the current receiving branch |
| 1. Create feature1 branch 2. Checkout feature1 branch 3. Create new file and commit in feature1 branch 4. Checkout main branch 5. Create file and commit it in the main branch 6. Merge feature1 branch into the main branch 7. Delete feature1 branch | 1. >>>git branch feature1 2. >>>git switch feature1 3. >>>git add file & >>>git commit -m „Text“ 4. >>>git switch main 5. >>>git add file & >>>git commit -m „Text“ 6. >>>git merge feature1 7. >>>git branch -d feature1 |
| Show all existing branches:  Create new Branch:  Switch to a Branch:  SHORT – Create and switch to branch: | >>>git branch  >>>git branch <branch\_name>  >>>git switch <branch\_name>  >>>git switch -c <branch\_name> |
| **NOTE:** >>>git switch <branch\_name> is the same as >>>git checkout <branch\_name> | |

**Anmerkung1**:

* >>>git show „<hashcode>:datei.py“  
  Zeigt den Inhalt der Datei in dem ausgewählten Commit (hashcode)

**Anmerkung2**:

* Reicht es sensible Daten wie Tokens in einem json File abzulegen und diesen dann in .gitignore zu packen?  
  Oder wären diese Files z.B. trotzdem in GitHIB?
* Alternativ müsste ich über die Umgebungsvariablen gehen! Das sollte klappen.

Frage:

* Sollte ich die .gitignore selber auch ignorieren oder sollte das Part oft he Staging sein?

git restore datei = git checkout -- datei

|  |
| --- |
| Bearbeiten von commits: zum Beispiel die Emil Adresse neu setzen nachdem man sie geändert hat:  [git - How to change the commit author for one specific commit? - Stack Overflow](https://stackoverflow.com/questions/3042437/how-to-change-the-commit-author-for-one-specific-commit)   1. git rebase -i --root 🡪 Alle commits ab root einzeln durchgehen 2. git commit --amend --reset-author 🡪 jeweils die Email Adresse neu setzen 3. git rebase --continue 🡪 Hiermit gehe ich immer ein commit weiter. Also für jeden commit die Zeile aus 2 ausführen |
| git lg - A better way to display your git logs in your console. Copy and paste this command into any project you're using Git with and you'll see a much nicer commit tree than the standard git reflog or git log:  git log --topo-order --all --graph --date=local --pretty=format:'%C(green)%h%C(reset) %><(55,trunc)%s%C(red)%d%C(reset) %C(blue)[%an]%C(reset) %C(yellow)%ad%C(reset)%n'  Or you can add this to your .gitconfig on your computer. This file is usually found at ~/.gitconfig:  [user]  name = Your Name  email = your.name@email.com  [alias]  lg = log --topo-order --all --graph --date=local --pretty=format:'%C(green)%h%C(reset) %><(55,trunc)%s%C(red)%d%C(reset) %C(blue)[%an]%C(reset) %C(yellow)%ad%C(reset)%n' |
|  |