GIT AND GITHUB BASICS

Git Configurations

git config --global user.name “<name>”
sets the name desired to be attached to your commit transactions

git config --global user.email “<email address>”
set the email address to be attached to your commit transactions

git config --global color.ui “<auto, true, or false>”
enables helpful colorization of command line output

git config —-global core.editor “<name of editor>”
git config --global core.editor.path “<editor path>”
sets the default editor

git config --global merge.tool “<name of merge tool>”
git config --global mergetool.path “<merge tool path with forward slashes>”
git config --global mergetool.prompt “<true or false>”
sets the mergetool editor

git config --global diff.tool “<name of diff tool>”
git config --global difftool.path “<diff tool path with forward slashes>”
git config --global difftool.prompt “<true or false>”
sets the difftool editor

git config --global push.default current
whenever you create a new local branch, a remote branch will always be created. Afterwards use ‘git push’ to the current branch of the same name (its upstream branch).

git config —-global push.default matching
whenever you use ‘git push’, Git will push local branches to remote branches that already exists with the same name.

git config —-global —-list
list all Git global configuration settings

git config —-list
list all Git configuration settings

git config —-global -e
list global configurations in the default editor

Git Aliases Configurations

```
git config –-global alias.br branch
git config –-global alias.co checkout
git config –-global alias.ci commit
git config –-global alias.st “status -s -b”
git config –-global alias.sta status
git config –-global alias.last “log -1 HEAD”
git config –-global alias.unstage “reset HEAD --”
git config –-global alias.log2 “log –-pretty=format: ‘%h %s [%an]’ –-graph”
git config –-global alias.logg “log –-graph –-decorate –-pretty=online –-
abrev-commit –-all”
git config –-global alias.logging “log –-oneline –-graph –-decorate
git config –-global alias.history “log –-all –-graph –-decorate –-oneline
git config –-global alias.history2color “log –-pretty=format: ‘%Cgreen%h
%Creset%ai | %s %Cblue[%an] %Cred%d’ –-date=short -n 10 –-
coloralias.graph=log –-graph –-decorate –-pretty=oneline –-abbrev-commit –-
all”
git config –-global alias.who2blame “log –-graph –-pretty=format: ‘%h %ad-
%s[%an]’”
```

Git Initialization

```
To start – create a new folder or directory
from the command line -

echo “#Some-new app” >> README.md
git init
git add README.md

git commit -m “Initial commit”
git branch -M main
git remote add origin “<https://github.com/username/repositiory_name>”
git push -u origin main

Or to push an existing repository
from the command line -

git remote add origin “<https://github.com/username/repositiory_name>”
git branch -M main
git push -u origin main
```

Git Clone Repository

```
git clone “<https://github.com/username/repositiory_name>”
creates a directory name of the git repository, initializes a .git folder
```

inside it, pulls down all the data for the repository, and checks out a working copy of the latest version of the repository.

git clone “<https://github.com/username/repositiory_name>” “<folder_name>” does the same thing as the above git clone command, but the target folder is the “<folder_name>”.

Permanently Stop Tracking A File

(If a file is already tracked by Git, .gitignore file does not apply. Git will continue to track changes to that file)

1. Add the file to .gitignore file

2. Run the following command
git rm —-cached “<fileName with extension>” -r

Git Workflow

git branch develop
git push -u origin develop

Branch develop will contain the complete history of the project, whereas branch main will contain a shorten version. Instead of branching off branch main, feature branches will branch develop as their parent branch. When feature branch is complete, it gets merged back into branch develop. Feature branches should never interact directly with branch main.

git checkout develop
git checkout -b feature_branch

When done with the development work on the feature branch, the next step is to merge the feature_branch into branch develop.

git checkout develop
git merge feature_branch

Once branch develop has acquired enough features for a release. A release version/1.X.X is merged into branch main. Then, to finish a release branch, use the following:

git checkout develop
git checkout -b release/1.X.X

git checkout main
git merge release/1.X.X

```
git tag -a v1.X.X -m “version 1.X.X”
git branch -D release/1.X.X
```

Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches, except they are based on the branch main instead of the branch develop. Hotfix branches are the only branches made directly from branch main.

```
git checkout main
git checkout -b hotfix_branch
```

When finishing a hotfix branch, it gets merged into both branch main and branch develop.

```
git checkout main
git merge hotfix_branch
git tag -a v1.X.X -m “version 1.X.X”
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

Example of Complete Gitflow:

```
git checkout main
git checkout -b develop
git checkout -b feature_branch
#Work happens on feature_branch

git checkout develop
git merge feature_branch
git checkout main
git merge develop
git branch -d feature_branch
#When ready for releases

git checkout develop
git branch release/1.X.X
git checkout main
git merge release/1.X.X
git git -a v1.X.X -m “version 1.X.X”
git branch -D release/1.X.X
```

Example of Complete Gitflow Hotfix

```
git checkout main
git checkout -b hotfix_branch
```

```
#Work is completed on hotfix_branch

git checkout develop
git merge hotfix_branch
git checkout main
git merge hotfix_branch
git checkout -b release/1.X.X
git tag -a v1.X.X -m “version 1.X.X”
git branch -D release/1.X.X
git branch -D hotfix_branch
```

Git Pull Request

1. On GitHub, fork the repository into your GitHub account.

2. Create a local copy on your computer and create a branch.
From the command line -

```
git clone “<https://github.com/username/repository_name>”
cd “<repository_name>”
git checkout -b “<pull_request_branch>”
```

3. Make changes to the pull_request_branch.

```
git add “<changes_made>”
git commit -m “<update: change_made>”
git push --set-upstream “<pull_request_branch>”
```

4. Create a pull request

Our changes are now uploaded to our remote Git repository. We are ready to make a pull request. To do so, go to the homepage of the forked version of the repository.

There is a button which says, “New pull request” that is associated with our new branch. When this is pressed, we create a pull request with our changes.

In the form we specify the name of our pull request (which is automatically set as a commit message of your most recent changes) and the description for our pull request.

Workflow For Changing A File Or Folder

1. Update a file or folder in the Git repository

2. Use git add to add those changes to the staging area

3. Use git commit to move changes from the staging area to a commit

4. Use git push to move/add changes to the main repository

Git Add

git add “<filename and extension>”
adds filename to the staging area

git add --all
git add .
adds every change made to files and folders from our repository to the Git staging area

Git Branch

git branch --list
git branch
lists all the branches in the repository

git branch -a
lists all branches both locally and remote

git branch -av
lists all branches (local and remote) with their last commit

git branch “<branch_name>”
creates a new branch with the name “<branch_name>”

git branch -d “<branch_name>”
deletes local ‘<branch_name>’. If there are unmerged changes, Git does not allow you to delete it.

git branch -D “<branch_name>”
forces delete local ‘<branch_name>’, even if there are unmerged changes. When you are sure to delete local ‘<branch_name>’ permanently, execute this command.

git push origin -delete <branch_name>
deletes the <branch_name> from GitHub

git branch -m “<branch_name>”
moves or renames the current branch to ‘<branch_name>’.

git branch -m “<branch_name>” “<new_branch_name>”

renames ‘<branch_name>’ to ‘<new_branch_name>’

git branch -r
displays the remote branches

git branch --verbose
git branch -v
shows sha1 and commit subject line for each HEAD, along with relationship to upstream branch (if any).

Git Checkout

git checkout “<branch_name>”
switches to the specified ‘<branch_name>’

git checkout -b “<branch_name>”
git switch -c “<branch_name>”
creates a new branch name ‘<branch_name>’ base off the current branch and directly switches to ‘<branch_name>’.

git checkout -b “<branch_name>” “<desired_branch_name>”
creates a branch based off ‘<desired_branch_name>’, and directly switches to ‘<branch_name>’

git checkout “<commit-hash>”
view a previous ‘<commit-hash>’ in DETACHED HEAD STATE - making commits in DETACHED HEAD STATE will not be associated with any branch. When leaving DETACHED HEAD STATE and returning to a branch, the commits made in this state are lost. They are not connected to any branch. To re-attach DETACHED HEAD - git switch “<branch_name>” or git checkout “<branch_name>” or git switch -

git checkout HEAD “<file_name>”
reset contents back to last commit

Git Clean

git clean -n
shows which files to be removed from the working directory

git clean --force
git clean -f
executes the clean command; that is, it removes the mentioned files shown in the ‘git clean -n’ command

Git Clone

git clone “<remote-repository-url>”
clones a repository to local machine in the specified folder/directory

git clone “<remote-repository-url>” “<folder_name>”
clone a repository to local machine in the specified ‘<folder_name>’

git clone --branch “<branch_name>” “<remote-repository-url>”
git clone -b “<branch_name>” “<remote-repository-url>”
clones the specified ‘<branch_name>’ to local machine in the specified folder/directory

git clone --branch “<branch_name>” “<remote-repository-url>” “<folder>”
git clone -b “<branch_name>” “<remote-repository-url>” “<folder>”
clones the specified ‘<branch_name>’ to local machine in the specified ‘<folder>’.

Git Diff

git diff
displays any uncommitted changes since the last commit

git diff HEAD
displays the difference between working directory and last commit

git diff --cached
displays the difference between staged changes and last commit

git diff --staged HEAD
displays the difference between what is staged and last commit on working branch

git difftool --stage HEAD
displays the differences in the configured difftool what is staged and the last commit on the working branch

git diff HEAD
displays differences since the last commit

git difftool HEAD
displays the differences since the last commit in the configured difftool

git diff --staged “<the file name>”
displays the differences between what is staged in ‘<the file name>’ and last commit on the working branch ‘<the file name>’

git difftool --stage “<the file name>”
displays the differences in configured difftool what is stage in ‘<the file name>’ and last commit on the working branch ‘<the file name>’

git diff ‘<commit-hash>’ HEAD
displays the difference between ‘<commit-hash>’ and the last commit

git diff HEAD HEAD^
displays the difference between last commit and the last commit minus 1

git diff “<commit-hash 1>” “<commit-hash 2>”
displays the differences between the ‘<commit-hash 1>’ and ‘<commit-hash 2>’ *note - use ‘ctrl q’ to cycle through the files and ‘q’ to escape

git diff main origin/main
displays the differences between local branch main and the remote branch main

git diff --name-only
shows only names of changed files

Git Fetch

git fetch
retrieves commits, files, and references from the remote repository into the local repository. Does not change your working state (note: It is harmless). It is more like checking to see if there are any changes available (note: It fetches but does not merge).

git fetch --dry-run
shows what would be done, without making any changes

Git Log

git log
displays a commit-hash number, the developer who made commit, the date the commit occurred, and information about the commit (a brief summary).

git log --author=”<author_name>”
searches for commits by a particular ‘<author_name>’. If found, returns commits by ‘<author_name>’; otherwise, nothing.

git log --grep=”<a pattern>”
searches for ‘<a pattern>’ and display only commits that have a message that matches ‘<a pattern>’.

git log --decorate
displays names of branches or tags of commits

git log --graph
displays a drawn text based graph of the commit on the left side of commit messages

git log -"<number>"
displays a limit of commits by the limited '<number>'

git log --no-merge
will not display merged commits (note: by default merge commit are listed).

git log --oneline
displays a condense version of each commit to a single line.

git log -p
displays the full diff for each commit.

git log --"<since>" --"<until>"
displays commits that occur between '<since>' and '<until>'. Arguments can be a commit ID, branch name, HEAD, or any other king of revision reference (note: the --since and --until flags are synonymous with --after and --before, respectively).

git log --stat
displays which files were altered and the relative number of lines that were added or deleted from each of them.

Git Merge

git merge "<branch_name>"
executes combining '<branch_name>' into the receiving current branch. It is used to combine multiple commits into one history.

git merge "<branch_name>" --no -ff
adds a merge commit to tracking

git merge "<branch_name>" -m "the message"
adds a message to merge commit

Git Pull

git pull

updates your local working branch with commits from the remote, and update all remote tracking branches (note: git pull is shorthand for git fetch, followed by git merge)

git pull --rebase
updates your local working branch with commits from the remote, but rewrite history so any local commits occur after all new commit coming from the remote, avoiding a merge commit (note: git pull --rebase is shorthand for git fetch, followed by git rebase).

Git Push

git push “<remote>” “<branch_name>”
creates a local branch in the destinated repository; and/or pushes the specified branch to along with all the necessary commits and internal objects

git push -u “<remote>” “<branch_name>”
when pushing a branch for the first time, this command will configure the relationship between the remote and local repository, sot that you can use ‘git pull’ and ‘git push’ with no additional options in the future.

Git push -d “<remote>” “<branch_name>”
deletes ‘<remote>’ ‘<branch_name>’ on GitHub

git push “<remote>” --all
pushes all your local branches to the specified ‘<remote>’

git push “<remote>” --force
forces the git push even if it results in a non-fast forward merge. DO NOT USE THE --force FLAG, unless you are absolutely sure you know what you are doing.

git push “<remote>” --tags
tags are not automatically pushed when push a branch or use the --all flag. The --tags flag sends all of local tags to the remote repository.

Git Rebase

git rebase
the process of moving or combining a sequence of commits to a new base commit. Rebasing changes the base of your branch from one commit to another, making it appear as if you created your branch from a different commit (note: It is important to understand that even though the branch looks the same, it is composed of entirely new commits).

git rebase “<branch_name>”
use this command
-as an alternative to merging
-as a cleanup tool
-get much cleaner project history
-no unnecessary merge commits
-end resuls is a linear project history

do not use this command
-never rebase commits that have been shared with others
-do not rebase already pushed commits up to GitHub, unless you are positive no one is using those commits.

git rebase -i “<branch_name>”
interactively rebases current branch onto ‘<branch_name>’. Launches default editor to enter commands for how each commit will transferred to the ‘<branch_name>’.

git rebase -i HEAD~”<range_of_commits>”
-i.e. git rebase -i HEAD~4
-running git rebase with the -i option will enter the interactive mode, which allows one to edit commits, add files, drop commits, etc. (note: that you will need to specify how far back you want to rewrite commits)
-notice that you are not rebasing onto another branch. Instead you are rebasing a series of commits onto the HEAD that they are currently based on.
-in the editor, you will see a list of commits alongside a list of commands to choose:
  -pick - use the commit
  -reword - use the commit, but edit the commit message
  -edit - use the commit, but stop for amending
  -fix up - use commit contents, but meld it into previous commit and discard the commit message
  -drop - remove commit

Git Reflog

git reflog
git reflog show HEAD
records updates made to the tip of the branch. It allows the return to commits even to the ones that are not referenced by any branch or tag. After rewriting the history, the reflog includes information about the previous state of branches and make it possible to go back to that state, if needed. In short, git reflog reference logs or ‘reflogs’ records when the tip of branches and other references were updated in the local repository.

-access commits that seem loast and not appearing in ‘git log’
remember:
git reset --hard <commit-sha>
git reflog show HEAD
git reset --hard <lost-commit-sha>

Git Remote

git remote remove “<remote_name>”
removes remote ‘<remote_name>’ (note: Usually the remote name is ‘origin’)

git remote add “<remote_name>”
adds remote ‘<remote_name>’ (note: Usually the remote name is ‘origin’)

Git Reset

git reset
resets staging area to match most recent commit, but leaving the working directory/folder unchanged.

git reset “<file_name>”
removes the ‘<file_name>’ from the staging area, but leaves the working directory unchanged (note: This unstages a file without overwriting any changes).

Git Restore

git restore “<file_name>”
restores ‘<file_name>’ to last commit

git restore --source HEAD~1 “<file_name>”
restores the contents of ‘<file_name>’ to its state from commit prior to HEAD

git restore --staged “<file_name>”
removes the ‘<file_name>’ from the staging area, but leaves its actual modifications untouched. By default git restore command will discard any local, uncommitted changes in corresponding files, and thereby restore their las committed state. With the --staged option, however, the ‘<file_name>’ will be only removed from the staging area. But its actual modifications will remain untouched.

Git Revert

git revert “<commit-sha>”

creates a brand new commit which reverses/undos the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

Git Shortlog

git shortlog
displays each author name, followed by number of commits, and list of commits

git shortlog -s
displays number of commits for each author

git shortlog -"<number>"
displays the limit of the last '<number>' of commits for each author

Git Stash

git stash
stores the changes to the working directory locally and allows one to retrieve the changes when needed (note: a handy command when switching between contexts).

git stash --all
git stash -a
stores the changes, untracked files, and ignored files to the working directory locally and allows one to retrieve the changes when needed. In short, add changes to ignored files (note: git stash will not stash new files that have not been staged and ones that have been ignored).

git stash --patch
git stash -p
stores the changes to specific files. For example, .gitignore file.

git stash --include-untracked
git stash -u
stores the changes and untracked files to the working directory locally and allows one to retrieve the changes when needed. This option includes to stash untracked files (note: git stash will not stash new files that have not been staged and ones that have been ignored).

git stash apply stash@{0}
git stash apply
applies the most recent stash without removing it from the stash. This is useful, if one wants to apply stashed changes to multiple branches. Git

assumes one wants to apply the most recent stash. This command reapplies the changes and keep them in the stash.

git stash apply stash@{stash_id}
applies a particular stash from the stash list to the current working directory

git stash clear
empties the stash list by removing all the stashes

git stash drop stash@{stash_id}
deletes a particular stash from the stash list

git stash list
displays the list of stashes

git stash pop stash@{0}
git stash pop
reapplies the changes to the current working directory and removes them from the stash list (note: this command is preferred, if one does not need the stashed changes to be reapplied more than once).

git stash create “<stash_name>”
creates a stash entry and returns its object name without pushing it to the stash reflog

git stash show -p | git apply -R
un-applies the most recent stash

Git Status

git status
displays the current state of your working directory and staging area

git status --short
git status -s
displays the current state of your working directory and staging in a short format

git status --branch
git status -b
displays the current branch and tracking information in short format

git status --verbose
git status -v

displays the name of files that have been changed, and shows the textual changes that are staged to be committed (i.e., like the output of ‘git diff --cached’). If -v flag is specified twice, it also displays the changes int the working tree that have been staged (i.e., like the output of ‘git diff’).

Git Switch

git switch “<branch_name>”
switches to ‘<branch_name>’ and makes it the current ‘HEAD’ branch. This command provides a simpler alternative to the classic ‘git checkout’ command.

git switch -c “<branch_name>”
creates a new branch named ‘<branch_name>’ and switches to it

git switch “<branch_name>” --discard-changes
switches to the specified ‘<branch_name>’ and discard any local changes to obtain a clean working copy.

git switch -
switches back to the previous checked out branch

Git Tag

git tag
tags are pointers that refer to a particular points in Git history. Tags are most often used to mark version releases in a repository. One can think of tags as branch references that do not change (a label for a commit)

git tag --list
git tag -l
displays a list of all tags

git tag --list “*<version_name>”
git tag -l “*<version_name>”
displays a list of all tags with ‘<version_name>’ afterwards

git tag --list “<version_name>*”
git tag -l “<version_name>*”
displays a list of all tags with ‘<version_name>’ before

git tag --list “*<version_name>*”
git tag -l “*<version_name>*”

displays a list of all tag that contains ‘<version_name>’. The git diff command can be use (note: git diff v17.0.0 v17.0.1) to view the difference between to versions

git tag “<tag_name>”
makes an unsigned tag locally. This is a lightweight tag. By default, Git will create the tag referring to the commit that HEAD is referencing.

git tag --delete “<tag_name>”
git tag -d “<tag_name>”
deletes a local tag

git tag -a “<tag_name>”
adds an unsigned, annotated tag (adds a label). Git will open your default text editor, and prompt you for addition information.

git tag “<tag_name>” -m “<message>”
adds a tag with a ‘<message>’. Passes a message directly and foregoes opening the default text editor.

git push “<remote>” “<tag_name>”
pushes the tag to GitHub

git push “<remote>” “<branch_name>” --tags
pushes all the tags to GitHub

git push “<remote>” “<branch_name>” “<tag_name>”
pushes ‘<branch_name>’ to GitHub

git push --delete “<remote>” “<tag_name>”
git push -d “<remote>” “<tag_name>”
git push “<remote>”:”<tag_name>”
deletes a ‘<remote>’ tag from GitHub

git show “<tag_name>”
displays the metadata about the tag

git tag “<tag_name>” “<commit_sha>”
makes a ‘<tag_name>’ of ‘<commit_sha>’

git tag “<tag_name>” -f
updates ‘<tag_name>’ with force