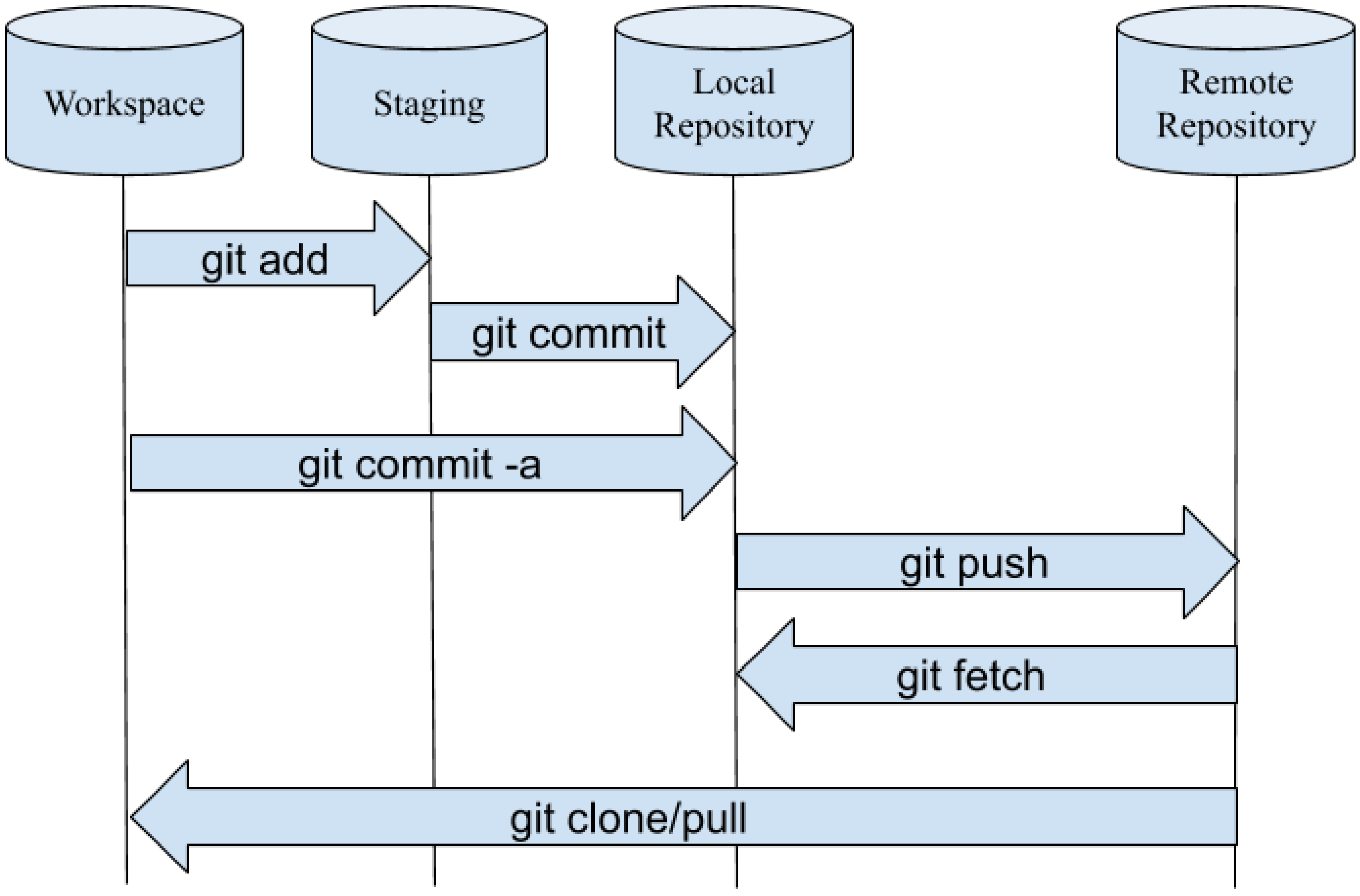
M. Mücahid Benlioğlu
Mert Koşan

Git Tutorial

Version control system that tracks the changes in files and helps group of people to work together in a project easily.

Architecture



Note: *Workspace is what you see in your folder. Staging area is changes that you marked as ready to be committed. Local repository contains snapshots of staging area called commits. Remote repository is more less a mirror of the local repository that you sync.*

Commands

Some commands, theirs usage and practice are retrieved from
http://guides.beanstalkapp.com/version-control/common-git-commands.html

We are also introducing some missing part of above link. We are going to share the codes and commands which we have done in the workshop as well.

git init

Marks a directory as a *"git repository"*. This is the first thing you need to do before using any commands

Usage:

```
# change directory to codebase
$ cd /file/path/to/code

# make directory a git repository
$ git init
```

Practice:

```
# change directory to codebase
$ cd /Users/computer-name/Documents/website

# make directory a git repository
$ git init
Initialized empty Git repository in /Users/computer-name/Documents/website/.git/
```

git add

Add changes to staging area from working directory for Git. Changes can be creating new/renaming/moving a file or directory, adding/removing/modifying some part of a file or changing permissions of a file/directory. Changes that will be committed, should be added to staging area first.

Usage:

```
$ git add <file or directory name>
```

Practice:

```
# Add all files and folders in the current directory
# Do this in the root of your project to add ALL the
# files and directories within your project
$ git add .

# Add a specific file or folder by replacing . with their realtive path
$ git add index.html
$ git add css/style.css
$ git add js/
```

git commit

Take a snapshot of changes that are added to the staging area with `git add` to the local repository. Each commit has own unique ID. You're required to write a commit message that describes the changes included that commit

Usage:

```
# Adding a commit with message
$ git commit -m "Commit message in quotes"
```

Practice:

```
$ git commit -m "My first commit message"
[SecretTesting 0254c3d] My first commit message
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 homepage/index.html
```

git diff

Show the difference between the areas (commits, working directory, staging area etc.).

Usage:

```
# Shows the difference between workspace and staging
$ git diff

# Shows the difference between workspace and commit
$ git diff HEAD

# Shows the difference between staging and commit
$ git diff --cached
```

Practice:

```
# Show difference between workspace and staging area
$ git diff
diff --git a/file b/file
index 8baef1b..5f5521f 100644
--- a/file
+++ b/file
@@ -1 +1,2 @@
 abc
+def
```

git status

Returns the current state of the repository. It shows untracked files or files which are ready to commit. If there are no files like these, it shows working directory is clean.

Usage:

```
$ git status
```

Practice:

```
# Message when files have not been staged (git add)
$ git status
On branch SecretTesting
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        homepage/index.html

# Message when files have been not been committed (git commit)
$ git status
On branch SecretTesting
Your branch is up-to-date with 'origin/SecretTesting'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   homepage/index.html

# Message when all files have been staged and committed
$ git status
On branch SecretTesting
nothing to commit, working directory clean
```

git config

There are many configuration that should or may be set in Git. There is a --global flag that set/get default settings for all repositories on your computer. Without a --global flag, git config applies only current repository. Two essential configurations are username and email, you should set them before the first usage of git.

Usage:

```
$ git config <setting> <command>
```

Practice:

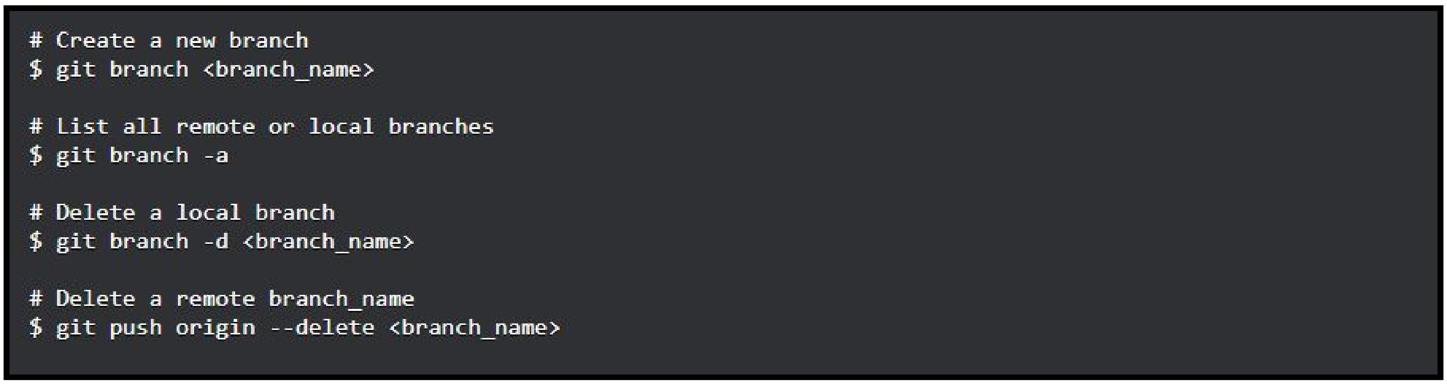
```
# Running git config globally
$ git config --global user.email "my@emailaddress.com"
$ git config --global user.name "Brian Kerr"

# Running git config on the current repository settings
$ git config user.email "my@emailaddress.com"
$ git config user.name "Brian Kerr"
```

git branch

In Git, there can be other branches alongside of the master branch. This command shows current branch where you are working on, creates or deletes branches.

Usage:



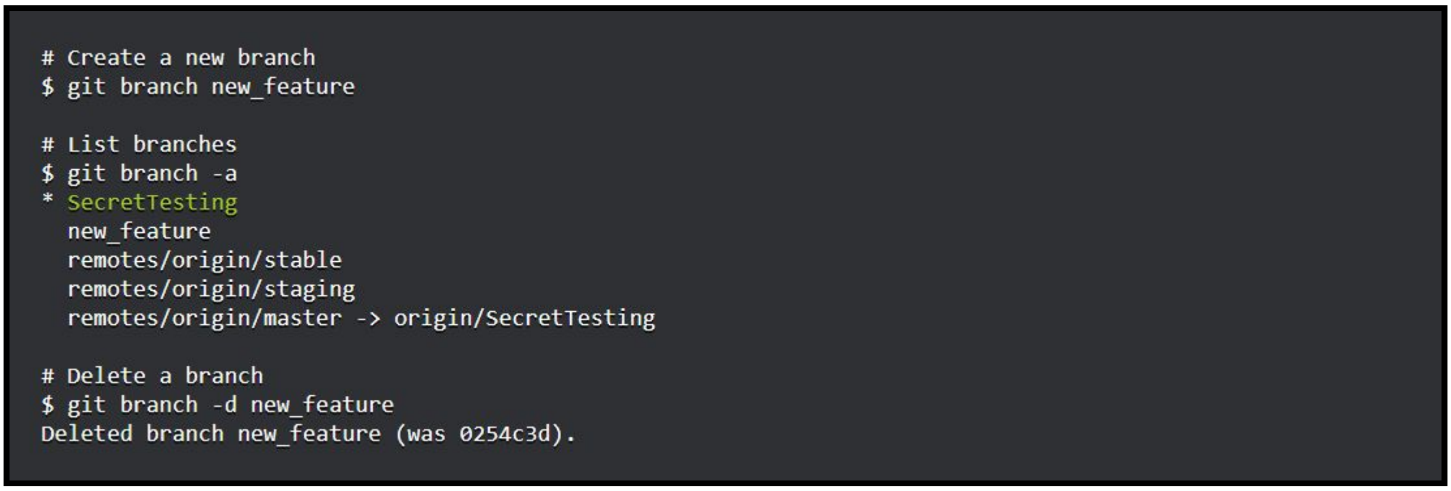
```
# Create a new branch
$ git branch <branch_name>

# List all remote or local branches
$ git branch -a

# Delete a local branch
$ git branch -d <branch_name>

# Delete a remote branch_name
$ git push origin --delete <branch_name>
```

Practice:



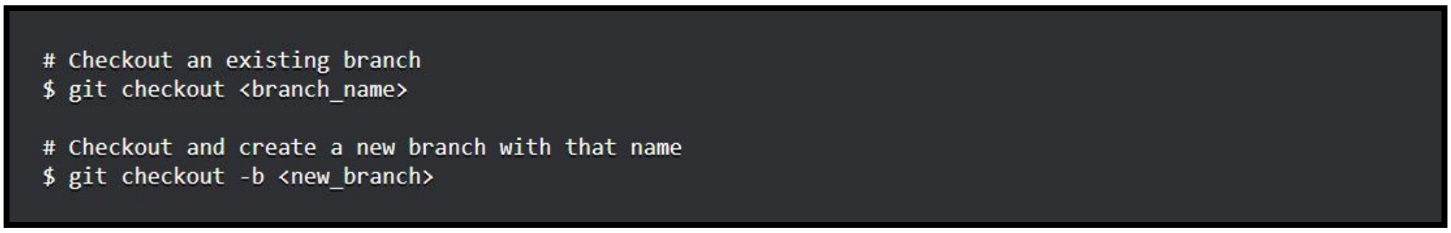
```
# Create a new branch
$ git branch new_feature

# List branches
$ git branch -a
* SecretTesting
  new_feature
  remotes/origin/stable
  remotes/origin/staging
  remotes/origin/master -> origin/SecretTesting

# Delete a branch
$ git branch -d new_feature
Deleted branch new_feature (was 0254c3d).
```

git checkout
Switches branches or restore files from repository

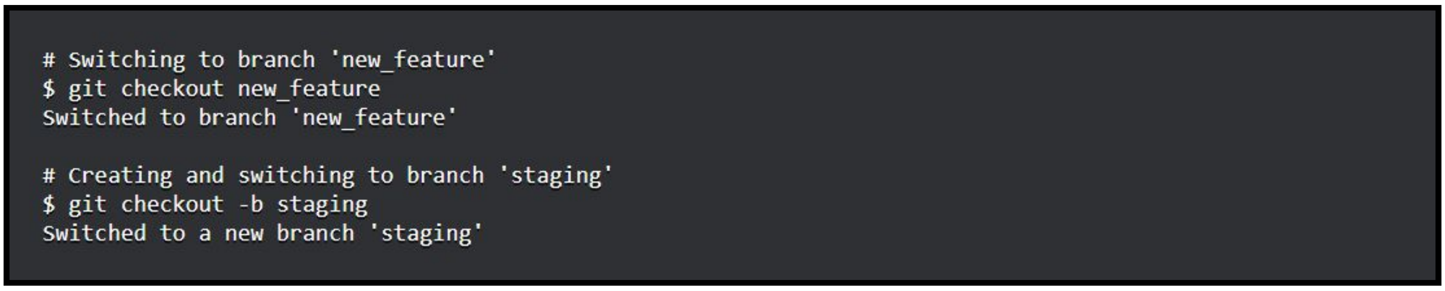
Usage:



```
# Checkout an existing branch
$ git checkout <branch_name>

# Checkout and create a new branch with that name
$ git checkout -b <new_branch>
```

Practice:

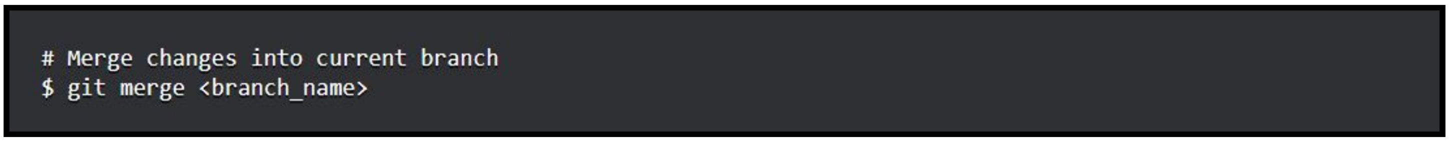


```
# Switching to branch 'new_feature'
$ git checkout new_feature
Switched to branch 'new_feature'

# Creating and switching to branch 'staging'
$ git checkout -b staging
Switched to a new branch 'staging'
```

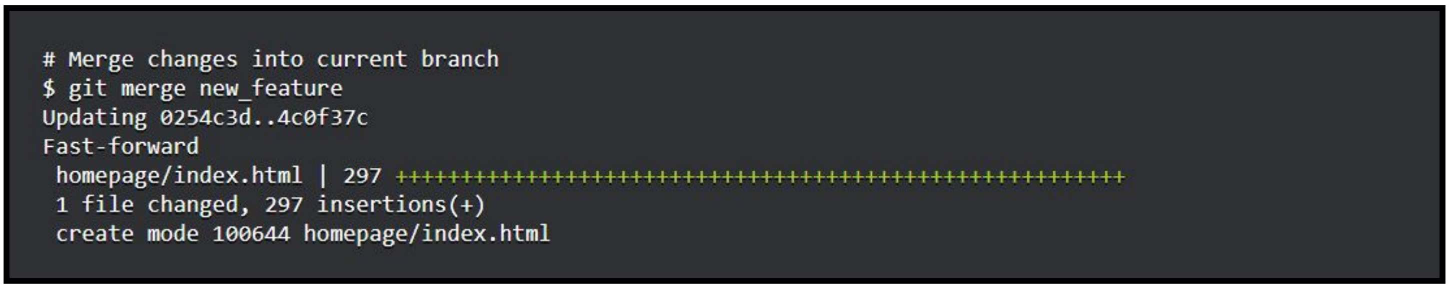
git merge
This combines the branches into one branch.

Usage:



```
# Merge changes into current branch
$ git merge <branch_name>
```

Practice:



```
# Merge changes into current branch
$ git merge new_feature
Updating 0254c3d..4c0f37c
Fast-forward
 homepage/index.html | 297 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 297 insertions(+)
 create mode 100644 homepage/index.html
```

Remote repository commands:

git remote:

It connects local repository to remote repository. A remote repository can take name that helps not to remember repository URL.

Usage:

```
# Add remote repository
$ git remote <command> <remote_name> <remote_URL>

# List named remote repositories
$ git remote -v
```

Practice:

```
# Adding a remote repository with the name of beanstalk
$ git remote add origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git

# List named remote repositories
$ git remote -v
origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (fetch)
origin git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git (push)
```

git clone:

It creates local copy of an available remote repository.

Usage:

```
$ git clone <remote_URL>
```

Practice:

```
$ git clone git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
Cloning into 'repository_name'...
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (5/5), 3.08 KiB | 0 bytes/s, done.
Checking connectivity... done.
```

git pull:

Gets the latest commits from remote repository and merges them in your branch.
Usage:

```
$ git pull <remote_URL/remote_name> <branch_name>
```

Practice:

```
# Pull from named remote
$ git pull origin staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.

# Pull from URL (not frequently used)
$ git pull git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git staging
From account_name.git.beanstalkapp.com:/account_name/repository_name
 * branch            staging    -> FETCH_HEAD
 * [new branch]      staging    -> origin/staging
Already up-to-date.
```

git push:

Sends local changes/commits to remote repository. Needs branch name and remote name as parameters.
Usage:

```
$ git push <remote_URL/remote_name> <branch>

# Push all local branches to remote repository
$ git push –all
```

Practice:

```
# Push a specific branch to a remote with named remote
$ git push origin staging
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 734 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
   ad189cb..0254c3d  SecretTesting -> SecretTesting

# Push all local branches to remote repository
$ git push --all
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 373 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To git@account_name.git.beanstalkapp.com:/acccount_name/repository_name.git
   0d56917..948ac97  master -> master
   ad189cb..0254c3d  SecretTesting -> SecretTesting
```

git fetch:

Gets the latest commits from remote repository, but unlike `pull` it doesn't merge them in your branch, You need to merge them manually if you want.

Usage:

```
# Fetch all data from remote repository to local repository
$ git fetch <remote_URL/remote_name>
```

Practice:

```
# Fetch data from remote called origin
$ git fetch origin
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3
Unpacking objects: 100% (3/3), done.
From https://github.com/cs-workshops/git-tutorial
* [new branch] GitIntro -> origin/GitIntro
```

git log:

Shows commit history for the current repository.

Usage:

```
# Show entire git log
$ git log

# Show git log with date pameters
$ git log --<after/before/since/until>=<date>

# Show git log based on commit author
$ git log --<author>="Author Name"
```

Practice

```
# Show entire git log
$ git log
commit 4c0f37c711623d20fc60b9cbcf393d515945952f
Author: Brian Kerr <my@emailaddress.com>
Date:   Tue Oct 25 17:46:11 2016 -0500

    Updating the wording of the homepage footer

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67
Author: Ashley Harpp <my@emailaddress.com>
Date:   Wed Oct 19 16:27:27 2016 -0500

    My first commit message

# Show git log with date pameters
$ git log --before="Oct 20"
commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67
Author: Ashley Harpp <my@emailaddress.com>
Date:   Wed Oct 19 16:27:27 2016 -0500

    My first commit message

# Show git log based on commit author
$ git log --author="Brian Kerr"
commit 4c0f37c711623d20fc60b9cbcf393d515945952f
Author: Brian Kerr <my@emailaddress.com>
Date:   Tue Oct 25 17:46:11 2016 -0500

    Updating the wording of the homepage footer
```

.gitignore :

.gitignore is a file, which tells git to ignore some files or folders that you don't want to track. Each line in a gitignore file specifies a pattern.

PATTERN FORMAT

- A blank line matches no files, so it can serve as a separator for readability.
- A line starting with # serves as a comment. Put a backslash ("`\`") in front of the first hash for patterns that begin with a hash.
- Trailing spaces are ignored unless they are quoted with backslash ("`\`").
- An optional prefix "`!`" which negates the pattern; any matching file excluded by a previous pattern will become included again. It is not possible to re-include a file if a parent directory of that file is excluded. Git doesn't list excluded directories for performance reasons, so any patterns on contained files have no effect, no matter where they are defined. Put a backslash ("`\`") in front of the first "`!`" for patterns that begin with a literal "`!`", for example, "`\!important!.txt`".
- If the pattern ends with a slash, it is removed for the purpose of the following description, but it would only find a match with a directory. In other words, `foo/` will match a directory `foo` and paths underneath it, but will not match a regular file or a symbolic link `foo` (this is consistent with the way how pathspec works in general in Git).
- If the pattern does not contain a slash /, Git treats it as a shell glob pattern and checks for a match against the pathname relative to the location of the `.gitignore` file (relative to the toplevel of the work tree if not from a `.gitignore` file).
- Otherwise, Git treats the pattern as a shell glob: "`*`" matches anything except "`/`", "`?`" matches any one character except "`/`" and "`[]`" matches one character in a selected range. See fnmatch(3) and the FNM_PATHNAME flag for a more detailed description.
- A leading slash matches the beginning of the pathname. For example, "/*.c" matches "cat-file.c" but not "mozilla-sha1/sha1.c".

Two consecutive asterisks ("`**`") in patterns matched against full pathname may have special meaning:

- A leading "`**`" followed by a slash means match in all directories. For example, "`**/foo`" matches file or directory "`foo`" anywhere, the same as pattern "`foo`". "`**/foo/bar`" matches file or directory "`bar`" anywhere that is directly under directory "`foo`".
- A trailing "`/**`" matches everything inside. For example, "`abc/**`" matches all files inside directory "`abc`", relative to the location of the `.gitignore` file, with infinite depth.
- A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "`a/**/b`" matches "`a/b`", "`a/x/b`", "`a/x/y/b`" and so on.
- Other consecutive asterisks are considered regular asterisks and will match according to the previous rules.

Best Practices

.gitignore : You should always have a .gitignore file in the root of your repository. This file tells git to completely ignore some files or folders. For example compiled binaries or large inputs that can be downloaded by each developer or personal setting files for IDEs etc. These files if not ignored will either take up huge unnecessary space, or will create problems with other team members by messing up their own personal settings. (**Hint:** you can use https://gitignore.io/ to get some mostly used language or IDE or OS specific gitignore files)

Regular commits: Don't wait until every single task is done to commit your changes. Divide your goal into relatively short sub-tasks that will take approx. 2-3 hours. (e.g. Implementing algorithm A, Reading input files and parsing them for further processing, Adding command line arguments etc.)

Commit messages: You should always write short but descriptive messages of what you have changed in that commit. Don't write meaningless comments (e.g. "blahblahblah", "some changes" or "some fixes" to every single commit) as they won't help you understand what changed or what task is done. A good commit message has a short description of what you did. (preferably 80-120 characters) followed by which files affected and how. You may use "*" for modifications, "+" for new files and "-" for removed files. Example:

```
Fixed this this this problem that happens when this this is done:
  * some/file.cpp:    Changed foo function to take that input
  + other/code.cpp:   Added new class to handle this and that functionality
  - oldfold/trash.py: Removed as someother class now does required work
```

Branches: This may seem unnecessary if you're working alone on a short homework, but it's especially important on team projects that have longer and more complex tasks. Your `master` branch should have the latest tested and stable version of your code. Optimally nobody should directly edit (push) to this branch, only tested changes should be added via a pull request. You can alternatively have a `development` branch where everyone adds their changes via pull request again and all changes later collectively tested and then merged to `master`. Everyone should work on their own branch for their assigned tasks, their branch name should preferably contain their username and title of task (e.g. `brucewayne/gps-bug`) This will isolate and make it easier to work on individual tasks.

Readme file: You should have Readme file in your project that has the project title, a short description, prerequisites to run this project, how to setup and run it etc. You can find many examples on the internet. Also you should use markdown syntax to write it, since it is supported by all big git repository hosts and it is a relatively easy way of formatting your text file.

A good IDE: Although it is very important to know how to use git commands from terminal, your life will be easier if you use a good IDE such as IntelliJ, PyCharm, CLion, Android Studio (my favourites) or Visual Studio or Xcode. A good IDE will help you handle hard situations that will take long time if done from terminal, easily.