**GIT**

Devops team – take the raw code (Ex: Java) from developers and convert into binary (like <.exe> file) and give it to testing team. (Testing team tests)

GIT is a version tool or Source Code Management Tool (SCM)

Central Repositories

Testing

Repositories

Devops into binaries

Developers

pull

push

Anywhere GIT is installed, it becomes a central repository even in local machine.

**Installation Steps:**

* Connect to an ec2 Instance
* <.pem> should be in same folder where git bash is done.
* **mkdir** git
* **Sudo yum install git**
* **git init** - command to initialise the respective directory as git repository

**how do you check a directory is git repo or not?**

* ls -a 🡪 if <.git> folder is present, then that current directory is git repo.

**The file goes through certain stages –**

1. **git add** <file>
2. **git commit -m “**message**”**
3. **git push**
4. **git status** 🡪 <Working tree clean> means all files are added in that working directory.

**Git Commands:**

* **vi** f1 🡪 add few lines
* **git add** f1
* **git commit -m** "message"
* **git status** --> details of file's status
* **git log** --> shows the commit ID's (history of the repository)
* **git config --global user.name** "pqr"
* **git config --global user.email** "pqr@gmail.com"
* **git checkout commitID** 🡪 switching to respective commitID
  + - touch f2 , f3
    - git add \* 🡪 <\*> if there are multiple files to add
    - git commit -m "f2 added"

**how to change the commit message ?**

* **git commit --amend -m** "New commit message."

**Branching**

* A branch in Git is a series of interrelated commits. When a repository is initialized in Git, a branch will be created by default. This default branch is called the **master**.
* Multiple branches can be created within a Git repository. When a developer starts working on a new feature of the project, he may create a new feature branch and work in isolation from the **master** branch. Once the feature is completed changes in that branch will be merged to the **master** branch. In other words, the **master** branch will be the main line of work. The **master** itself is a pointer to the latest commit.
* The **HEAD** is a reference that points to the **master**. Every time you commit, Git updates both **master** and the **HEAD** pointers to point to the last commit by default.

<https://www.tutorialspoint.com/what-is-the-difference-between-head-and-master-in-git>

* Whenever we follow the process of git add 🡪 git commit 🡪 the file is added to branch called master.
* Having parallel development on the same piece of code is called branching. After developing it can be merged or integrated with the master.
* it is for parallel development; teams can work on same piece of code on different branches parallels and later integrate by merging.
* If we do not “add and commit” files it appears in all branches
* only 1 master per repository

**why we need branching?** 🡪 for parallel development

* **git branch** 🡪 list all branches
* **\*** indicates we are on that branch
* **git branch <branch\_name>** 🡪 create new branch
* **git checkout <branch\_name>** 🡪 switch to branch
* **git branch -d <branch\_name>** 🡪 delete branch
* **git branch <branch2> < branch1>** 🡪 creates new branch2 under branch1
* **git branch <branch2>** 🡪 creates new branch under branch where it is checked out

**Tagging**

* It is name given to set of versions of files and directories. it is easy to remember the tag names, it indicates milestone of a project.
  + **git tag** 🡪 to list all tags
  + **git tag <tag\_name>** -- create tag
  + **git checkout <tag\_name>** -- switch to tag
  + **git tag -d <tag\_name>** -- delete tag

**Difference b/w tag and branch ?**

* tag is name given to set of versions of files and directories
* branching is for parallel development

**Merge**

* **git merge <branch\_name>** --- merge specified branch to checked out branch. create a new commit ID indicating merge
* Merging is only 1 way 🡪 from source to destination. **You cannot undo merging**
* Also, cannot check which branch is created under which branch

**What is merge conflict ?**

* Merge conflict occurs when same line of code is modified on 2 different branches . In this case we contact the person who changed the code on respective branches and once they let us know which changes has to be retained , we go with those changes .

**git log <file\_name>** 🡪 to trace back the developers

**Cherry-pick**

* **git cherry\_pick <commit\_id>** 🡪 to merge specific commitID with currently checked out branch
* **git cherry\_pick <commit\_id1> <commit\_id2>** 🡪 to merge specific commit id with another specific commit id (not entire branch or master)

**Rebase**

* Rewrites the history by creating new commits for each commit in source branch
* It is similar to merge.
* **git rebase <branch\_name>**

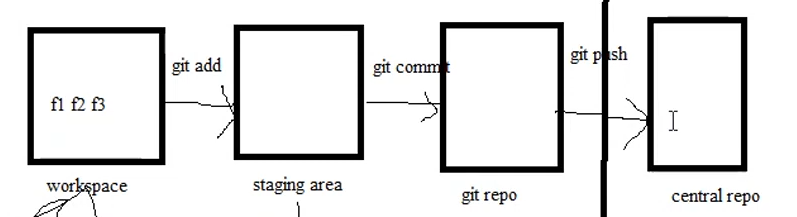
**What is the difference between Merge and rebase?**

* Both merge and rebase perform same operation of integrating branches, but the difference is how they do it.
* **Merge** : Creates new commitID indicating merge. Merge conflict can be handled easily, as the commits are reachable
* **Rebase** : Rewrites the history by creating new commits for each commit in source branch since commit history is rewritten, it will be difficult to understand the conflict in some cases as commits are no longer reachable.
* **Squash** : it is a technique to condense large number of commits to make into a small number of meaningful commits so that we can make git history clear
* **git rebase -i HEAD~n**

**GIT Architecture:**

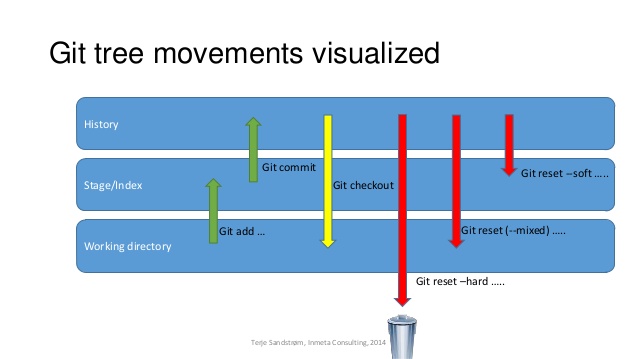
we have 4 stages

1. **Workspace** - it is a place where we edit , modify project related files. All the file in workspace is visible to all directories.
2. **Staging Area** - On **Git Add**, files are moved from workspace to staging area where changes are saved
3. **Local Repo** - on **Git Commit** , files will be added to local/git repo & then we can track the file versions. Commit ID are created here.
4. **Central repo** - On **Git Push**, files are moved to central repo.



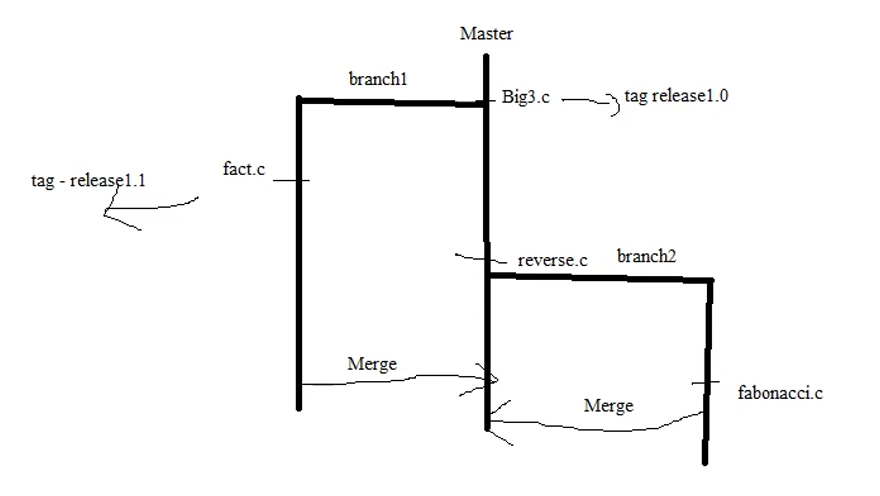
**Git Reset vs Revert:**

* **git reset** 🡪 undo the committed changes, history will be removed
* git reset --mixed / git reset HEAD 🡪 moves files from staging area to workspace
* git reset --soft <commitID> 🡪moves files from git repo to staging area, history will be removed
* git reset --hard <commitID> 🡪moves files from git repo, staging area, workspace and commitID's are also removed



* **git revert** 🡪 undo the committed changes , history will be retained, we can track the changes in git log
* git revert HEAD
* git revert commitID

**Assignment**:



**Git stash:**

* It locally stores all the most recent changes in a workspace and resets the state of the workspace to the prior commit state.
* A user can retrieve all files put into the stash with the **git stash pop** and **git stash apply** commands.
* Enables user to retain or not to lose edits. It stores in some temporary area.
* **git stash push** 🡪 Creates a new stash and rolls back the state of all modified files;
* **git stash pop** 🡪 Takes the files in a stash, places them back into the development workspace and deletes the stash from history;
* **git stash apply** 🡪 Takes the files in a stash and places them back into the development workspace, but does not delete the stash from history;
* **git stash list** 🡪 Displays the stash history in chronological order; and
* **git stash clear** 🡪 Removes all entries in the git stash history

**Git stash vs. commit:**

* The git commit and git stash commands are similar in that both take a snapshot of modified files in the git working tree and store that snapshot for future reference.
* The key differences between the two are as follows:

|  |  |
| --- | --- |
| git commit | git stash |
| commit is part of the public git history | stash is stored locally |
| commit creates a new save point on a branch | a stash reverts to a previous save point |
| A new commit leaves files in the working tree unchanged | a stash resets files in the working tree to the previous commit point |
| A commit is a public record of file changes | a stash is local |

**Git stash vs. reset**

* The git stash and the git reset hard commands are similar, as both commands will revert all files in the working directory back to the state at the point of the previous commit.
* Differences between the two include:

|  |  |
| --- | --- |
| git reset | git stash |
| A reset creates a new commit point in the branch history | stash does not |
| reset can jump back to any prior commit | stash can only reset the files in the workspace to the point of the previous commit |
| hard reset will discard all changes | stash saves a snapshot of all locally modified files |

**Git reset vs. revert**

|  |  |
| --- | --- |
| git reset | git revert |
| * a dangerous method * There is a real risk of losing work with git reset. Git reset will never delete a commit, however, commits can become 'orphaned' which means there is no direct path from a ref to access them. These orphaned commits can usually be found and restored using git reflog. Git will permanently delete any orphaned commits after it runs the internal garbage collector. By default, Git is configured to run the garbage collector every 30 days. Commit History is one of the 'three git trees' the other two, Staging Index and Working Directory are not as permanent as Commits. Care must be taken when using this tool, as it’s one of the only Git commands that have the potential to lose your work. | a “safe” way to undo changes |
| * git reset is designed to undo local changes to the Staging Index and Working Directory * resetting completely removes a changeset | * reverting is designed to safely undo a public commit * reverting maintains the original changeset and uses a new commit to apply the undo |
| * Make sure that you’re using git reset on a local experiment that went wrong—not on published changes | * If you need to fix a public commit, the git revert command was designed specifically for this purpose |

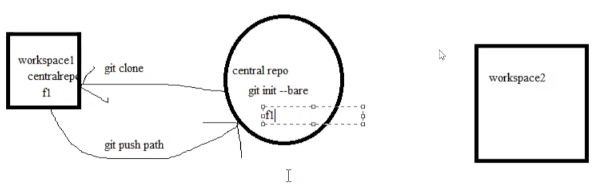
**Git stash vs. stage**

* The git stash and git stage commands can be confused because of their similar names, but the two commands are different. The git stage command simply adds files to the git index. This allows those files to be part of a filesystem snapshot when a git commit occurs. This is a different construct as compared to git stash.

**git checkout -- <filename>:** to undo uncommitted changes

**git checkout -- . :** to undo uncommitted changes for all files

**Central Repository:**

****

* **Git init --bare** 🡪 it initiates central repository. Here .git is not visible. Local rep commands don’t work here. Push, pull, clone, etc works
* **Git clone** **<URL>**🡪 initialises .git in central repo of workspace. No need to use git init

**Bare repo vs non-bare Repo**

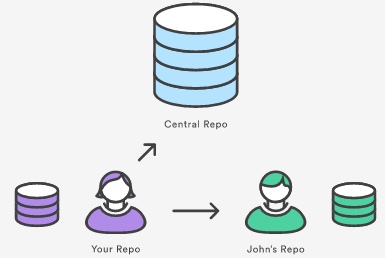
* bare repo – central repo only push and pull i.e., no “.git” folder,
* non-bare repo – local repo where “.git” is created

**Taking Remote code to Local (Fetch and Pull commands)**

* In a collaborative working environment, from time to time you would need to check if someone has committed any new changes to the project you’re working on, and update your working copy with those changes. This is called “Downstream”.
* There are two ways you can perform this task, one is a cautious way (the **fetch** command), other is a way of nonchalance (the **pull** command). You can think of both of these as the Download commands.
* **git remote add origin** [**https://github.com/bharathreddy06/practice.git**](https://github.com/bharathreddy06/practice.git)
* **git remote -v**
* **git push -u origin master**

**Remote**

* The git remote command lets you create, view, and delete connections to other repositories.
* Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.
* For example, the following diagram shows two remote connections from your repo into the central repo and another developer’s repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.

****

* **git remote** 🡪 List the remote connections you have to other repositories.
* **git remote -v** 🡪 above + includes the URL of each connection.

**Fetch**

* This command fetches the latest code commits from the remote and places them only in the local repository. The working directory and staging area remain unchanged.
* This option is for those who want to first check what is the latest code changes that are made by their fellow developers by checking the diff between their working copy and the local repository.
* Once they are satisfied with the changes, they will merge the local repository with the working copy using the merge command.

**git fetch** path(url)

**git checkout** FETCH\_HEAD

**git merge** FETCH\_HEAD

**Pull**

* This command fetches the latest code commits from the remote and merges with everything in all Three Trees of Git.
* You might end up in conflict after the Pull which you would need to resolve on your own.

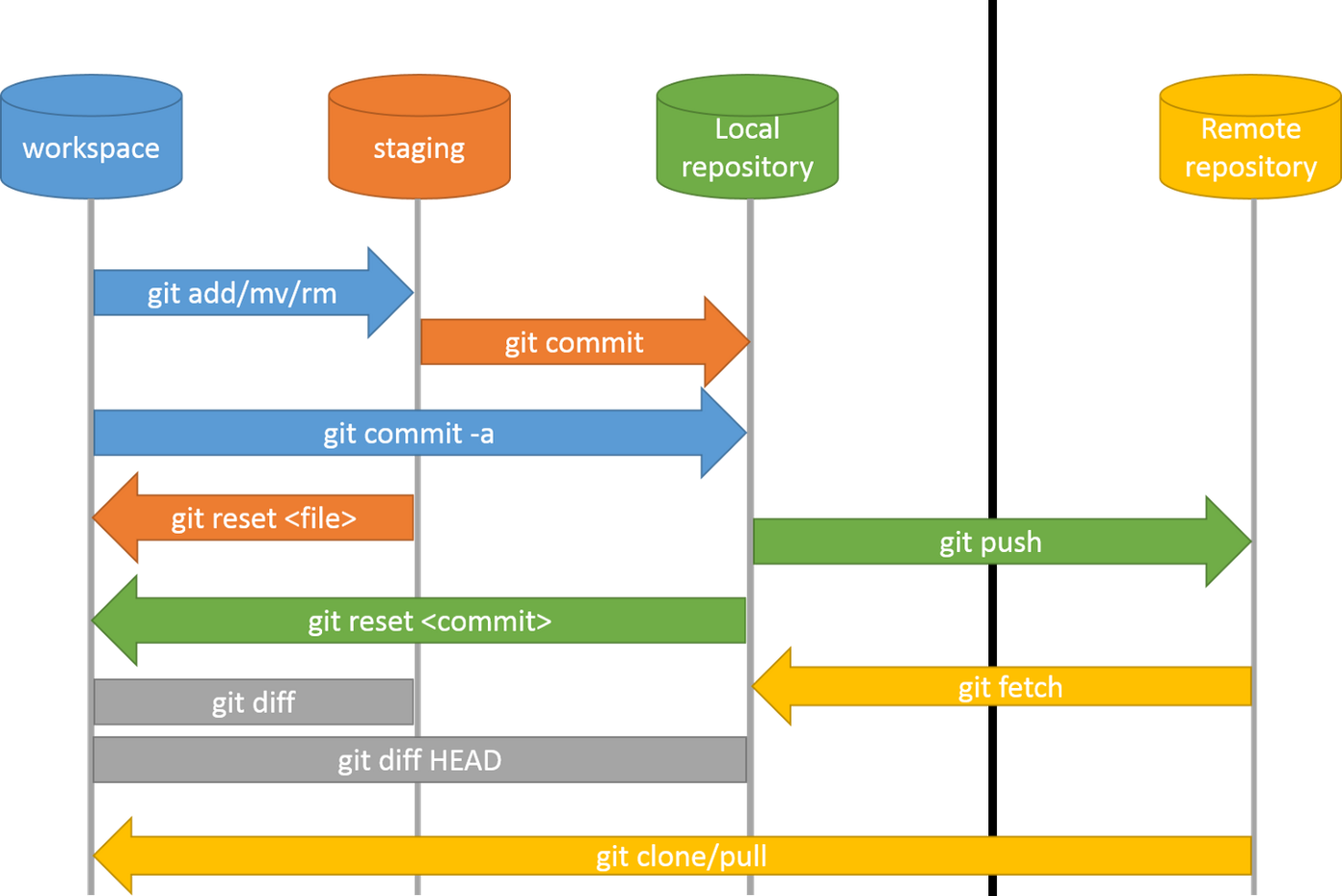
Pull = Fetch + Merge

**Clone:**

* If the local repository doesn’t exist, you can simply use the Clone command which will clone the remote repository on your machine and create a local repository and working directory for you.

**<.gitignore>:**

* The . gitignore file tells Git which files to ignore when committing your project to the GitHub repository.
* gitignore is located in the root directory of your repo.
* The .gitignore file itself is a plain text document.



**git vs git hub:**

* git is a local VCS (version control system) software that enables developers to save snapshots of their projects over time. It's generally best for individual use.
* GitHub is a web-based platform that incorporates git's version control features so they can be used collaboratively.

|  |  |
| --- | --- |
| **Git** | **GitHub** |
| Git is a software. | GitHub is a service. |
| Git is a command-line tool | GitHub is a graphical user interface |
| Git is installed locally on the system | GitHub is hosted on the web |
| Git is maintained by linux. | GitHub is maintained by Microsoft. |
| Git is focused on version control and code sharing. | GitHub is focused on centralized source code hosting. |
| Git is a version control system to manage source code history. | GitHub is a hosting service for Git repositories. |
|
| Git was first released in 2005. | GitHub was launched in 2008. |
|
| Git has no user management feature. | GitHub has a built-in user management feature. |
|
| Git is open-source licensed. | GitHub includes a free-tier and pay-for-use tier. |
| Git has minimal external tool configuration. | GitHub has an active marketplace for tool integration. |
| Git provides a Desktop interface named Git Gui. | GitHub provides a Desktop interface named GitHub Desktop. |
| Git competes with CVS, Azure DevOps Server, Subversion, Mercurial, etc. | GitHub competes with GitLab, Git Bucket, AWS Code Commit, etc. |

There are many **Version Control Systems** that are available in the market, but few top VCS are:

1. GIT
2. CVS
3. SVN
4. Assembla
5. Mercurial
6. Bazaar

**History of git commands:**

* **git show**: output in the same format as we studied in the git log with a difference of
  + The commit to which HEAD is pointing
  + Difference between the versions of the file to which HEAD is pointing
* **git log --since=<date>**: all commits including mentioned date
* **git log –oneline**: to see the list of commits.
* **git log --skip 4 –oneline**: skip 4 commits
* **git log --author=<name of author>**
* **git log –reverse**: reverse chronological order….old to latest

**git hook:**

* Git hooks are scripts that run automatically every time a particular event occurs in a Git repository. They let you customize Git's internal behaviour and trigger customizable actions at key points in the development life cycle.
* Present in .git/hook
* Executable and can be edited
* Pertained to local and not version controlled that is specific to user

**git alias:**

* It is important to note that there is no direct git alias command. Aliases are created through the use of the git config command and the Git configuration files. As with other configuration values, aliases can be created in a local or global scope.
  + $ git config --global alias.co checkout
  + $ git config --global alias.br branch
  + $ git config --global alias.ci commit
  + $ git config --global alias.st status
* globally stored shortcuts for common git commands
* Using aliases to create new Git commands

Ex: git config --global alias.unstage 'reset HEAD --' two commands equivalent.

git unstage fileA  
 $ git reset HEAD – fileA

* Aliases can be created through two primary methods:
  + Directly editing Git config files /.git/config
  + Using the git config to create aliases

**Difference Between git reset and git restore:**

* The restore command helps us to unstage or discard uncommitted local changes. It can be used for restoring files in the working tree from the index or another commit. This command does not update our branch. It can also be helpful to restore files in the index from another commit.
* Git reset is used for updating the current branch. It can also be helpful to restore the index, intersecting with git restore. Git reset is used for resetting the index to discard changes in the working tree. This command will not update our branch. It is mostly used with the --soft option, which only resets the index and leaves the working tree unchanged. It is useful if we have staged changes that we do not want to discard.
* Git reset will complete successfully if the working directory is clean (no changes to be committed), while the Git Restore will fail if the working directory is clean. Both can affect the HEAD. However, Git restore will only affect the HEAD indirectly through the staging area. Git reset can directly work with the index and the HEAD. Both can affect the HEAD. Git restore will only affect the index and the HEAD indirectly through the staging area.
* Git reset can be used to modify the local repository only if we have not pushed anything yet. If we have pushed to the remote server, git-reset will modify the staging area and working copy, but not the repository. It is useful if we want to undo local changes but do not want to reintroduce them again.
* Git restore is the opposite as it can only be used to modify our repository, not the staging area or local working copy. It will not affect any commits we have pushed.

**Git blame:**

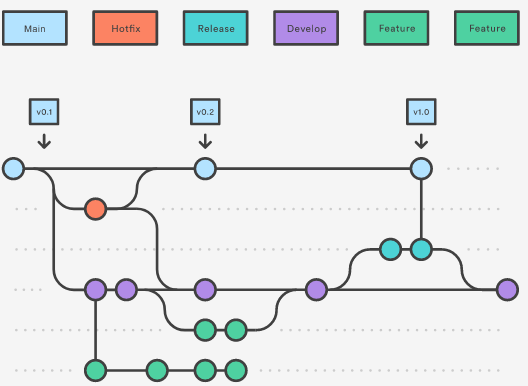
* The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was.

**Gitflow:**

* Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches.
* Gitflow is a legacy Git workflow, novel strategy for managing Git branches.
* Gitflow has fallen in popularity in favour of trunk-based workflows, which are now considered best practices for modern continuous software development and DevOps practices.
* Gitflow used for projects that have a scheduled release cycle and for the DevOps best practice of continuous delivery

**Hotfix branch**

* Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on main instead of develop. This is the only branch that should fork directly off of main. As soon as the fix is complete, it should be merged into both main and develop (or the current release branch), and main should be tagged with an updated version number.

****

**Difference Between fork and git clone**

|  |  |
| --- | --- |
| Fork | Clone |
| Forking is done on the GitHub Account | Cloning is done using Git |
| Forking a repository creates a copy of the original repository on our GitHub account | Cloning a repository creates a copy of the original repository on our local machine |
| Changes made to the forked repository can be merged with the original repository via a pull request | Changes made to the cloned repository cannot be merged with the original repository unless you are the collaborator or the owner of the repository |
| Forking is a concept | Cloning is a process |
| Forking is just containing a separate copy of the repository and there is no command involved | Cloning is done through the command ‘**git clone**‘ and it is a process of receiving all the code files to the local machine |

**Sources:**

* <https://toolsqa.com/git/dot-git-folder/>
* <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init>

**How to fork GitHub on the command line**

* <https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/command-line-GitHub-fork-CLI-terminal-shell>