Author: Ritesh Udhani

Please contact for queries

Installation (for Windows)

Git version 2.26.2 latest -20th April 2020

Git binary installs GUI and command line interface, Git Bash

The command line interface can use Linux commands and can also run from Windows command prompt

Installation from https://git-scm.com

Install P4 merge for diffing and merging. Install only the merge tool and not the other tools. (Optional but recommended)

Set up configurations as mentioned below

Documentation

Documentation at https://git-scm.com/doc

Reference for this document

Progit book on Git https://git-scm.com/book/en/v2

And other online resources

Git benefits:

Allows to create many branches without copying the entire source making it extremely fast to work in collaboration. Hence, allowing many workflows other than the centralized workflow of server and clients.

It is fast as it is doesn't use centralized server and because operations are performed locally

It is distributed and every clone has complete information with the history (Perforce, CVS, SVN were centralized systems) so we don't risk losing all the data. Also, many workflows can be created by collaborating with the non-central users.

Git stores snapshots and not differences like other VCS

It has staging area (index) not present in other versioning systems (along with a working directory and the compressed repository to which commits are done). The three states of a file are committed (shown by clean working directory in `git status`), modified or staged.

One downside is that though it is good for source code, it is not the best for large files, like video files, which are edited regularly.

Some definitions

Git: Version control system

GitHub: Most popular repository hosting service. Even source code of Git is on GitHub. Hosting public and private repositories is free. But hosting private repo with special features is paid.

Author: one who makes a patch

Committer: one who applies a patch to a repository

Repository: The files along with their history

Remote repository: version of your local repo which is held somewhere else over the internet (or on your PC itself)

Local repository: The folder with .git name is the local repo. This is different from working tree.

Working directory: Synonyms: Working tree, Workspace. The actual folder which contains the files that you edit. This folder also contains the .git folder i.e. the repository.

Staging area: A virtual area where the changes to be committed are held.

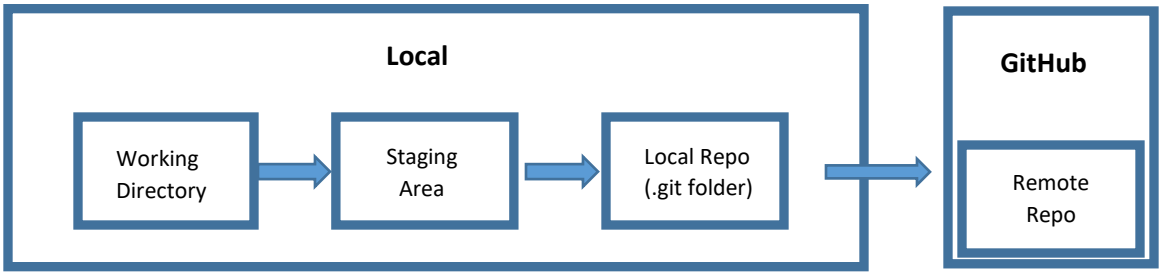

Figure 1: Git states with remote

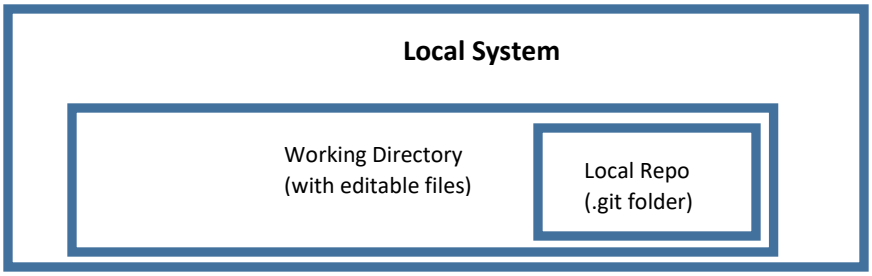

Figure 2: Physical structure of a local repo

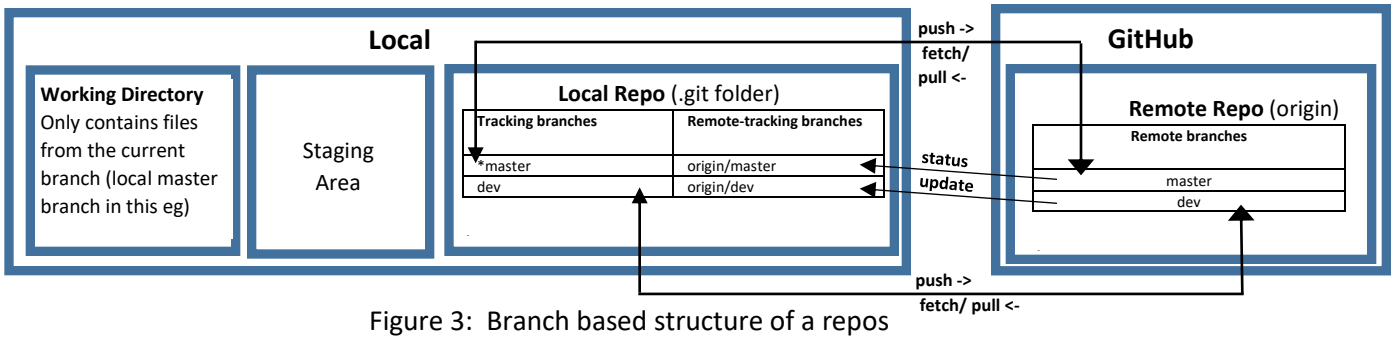

Figure 3: Branch based structure of a repos

Untracked files: Files present in working directory but not tracked by git

Tags: Tags are used to mark milestones in project development and are usually used for versioning of software. They are of two types: annotated and lightweight tags. The former one is recommended for tagging.

Commit checksum: Unique commit identifier

Commit object, blobs, and tree: Each commit is stored as a commit object containing information about the committer, author etc. And each file-snapshot in the commit is kept as blobs. A tree contains the information on which file belongs to which blob. And the commit object holds pointer to this tree and also pointer to previous commit objects

Branches: A branch is just a timeline of commits. A repository can have many branches. E.g. 'master'. A branch is a pointer to one of these commit objects. The branch pointer points to the latest commit object in that branch. Since a branch is only a label, deletion of a branch will not lead to deletion of any commit object.

Topic branches: Branches created temporarily for specific tasks which get deleted later after the small task is closed

Remote references: References to state of remote branches, remote tags and so on in the local repo. They are updated when the local repo is connected with the remote repo. They are of the form <remote repo>/<remote branch name>

Remote-tracking branch: they are of the form of origin/master and track the remote branches on a local repo.

Tracking branches: are branches that track a remote branch via a remote-tracking branch. The remote branch being tracked is known as the *upstream branch.* Short hand for upstream branch is @{u} or @{upstream} i.e. same as <remote repo>/<branch> e.g. 'origin/master'

HEAD: Generally points the last commit of the current branch. If it doesn't point to the last commit then it is termed as detached HEAD state. This happens when one checkouts a commit (or a tag) instead of the branch. Also, after a change is staged, the HEAD pointer points to these staged changes.

Merge types: fast-forward merge, automatic merge and manual merge.

Fast-forward merge: When the master branch (or other parent branch to which we merge) has no commit since the branch-to-be-merged was created, this type of merge occurs.

Automatic and manual merge: When master branch has diverged, i.e. has more commits, since the creation of the child branch these two merge types come into picture.

Stashing: If you are in the middle of a change in a working directory and you decide to make changes to another branch or change context in the same branch use stashing to stash your temporary work. This can also stash the code in staging area.

Rebasing: this technique is used instead of merging to apply new changes sometimes as it helps in generating a cleaner history. The end result of both commands can be the same. In rebasing, all the commits are applied again in the same order. However, in merging there is one final commit.

Pull request: Request to pull changes/patches (in your repo) from your branch to another branch. The command is 'git request-pull' and not 'git pull'.

Tag: Tags are used to version the source code. There are of 2 types: Lightweight and Annotated (Release has release notes in addition to tag information). GitHub can only create lightweight tags.

Markdown: GitHub uses markdown to create rich text. It can be used for various purposes like Readme.md, pull request, issue, conversation, etc. comments. File name extension is '.md'

Configuration files in increasing priority order

1. Location of gitconfig file for storing username and email, and other configurations for Git activities throughout the system for all users

```
C:\Program Files\Git\etc\gitconfig
```

2. Location of .gitconfig file for storing username and email, and other configurations for the user logged in

```
C:\Users\$USER\.gitconfig
```

3. Location of git config for the particular repo

```
.git\config
```

Gitignore file

.gitignore file keeps a record of those files which should not be tracked by Git in the working directory. Useful for binaries, libraries, etc. generated during builds which should not be committed to the repo. Please note that it is possible to have multiple .gitignore files for a project

A list of .gitignore files for various requirements is listed below at
https://github.com/github/gitignore

Configurations	Description

`git config --system user.name "Ritesh Udhani"`
`git config --global user.name "Ritesh Udhani"`
`git config --local user.name "Ritesh Udhani"`
	Sets username used for all commits.
	Updates the file #1, #2 and #3 above, respectively.

`git config --system user.email "****@gmail.com"`
`git config --global user.email "****@gmail.com"`
`git config --local user.email "****@gmail.com"`
	To set email address for all commits.
	Updates the file #1, #2 and #3 above, respectively.

`git config --list --show-origin`	To see from where all the configurations are picked. Listed in the order system, global and local, with increasing priority
`git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"` `git config --global core.editor emacs`	To set up your favorite text editor for git files. Notepad++ recommended for windows

`git config --global diff.tool p4merge`
`git config --global difftool.p4merge.path "C:\Program Files\Perforce\p4merge.exe"`
`git config --global difftool.prompt false`
	Enable p4merge as the default diff tool

`git config --global merge.tool p4merge`
`git config --global mergetool.p4merge.path "C:\Program Files\Perforce\p4merge.exe"`
`git config --global mergetool.prompt false`
	Enable p4merge as the default merge tool

Help on commands	**Description**
`git help <command>` `git <command> --help`	To get help on a command
Git basic commands	**Description**
Creating/getting a Git repository (repo)	
`git init`	To add version control to an existing folder
`git clone <project path>`	To clone a local repository from a remote repository. The .git folder is the repository and the files are a part of working directory (working tree).
`git init <new folder name>`	To create a new folder for your project and initialize a git repo in it.
Recording changes in a git repo	
`git add <filename>`	It means "add precisely this content to the next commit". This command is used to start tracking, staging and resolve merge conflicts
`git add *` `git add -u`	To add all the changed files to the staging area when we do not know what all files got changed
`git restore --staged <filename>` `git reset HEAD <filename>`	To unstage a file in the staging area (index)

`git restore <filename>`
 	
`git checkout -- <filename>`
	To discard modifications done in a file in working directory

`git status`	To check what is going on in the local repo. To check the status of each file whether it is modified, staged or up to date. It can also show if the file in the tracking branch is upto date with the remote-tracking branch
`git status -s`	For a simplified output
`git commit –m "message"`	Commit changes to the local repo
`git commit`	The commit message is to be entered through the default text editor for git
`git commit –a –m "message"`	To commit directly from working tree to the repository by skipping the staging area. Use very carefully as it can be risky.

`git rm <filename>`
is equivalent to
`rm <filename>`
 	
`git add –u`
	To remove an existing file from the repository. It also deletes the file from the working tree and also updates the staging area with this information. The second method asks to update git for the file deleted by rm command

`git checkout filename`	To restore a file deleted from the working directory using 'rm' or 'git rm'.

`git mv <filename1> <filename2>`
or
`mv <filename1> <filename2>`
 	
`git add –u`
 	
`git add <filename2>`
	To rename a file in git

Check differences	**Description**
`git diff`	To know the exact changes made in working directory but not staged. Compares staging area with working tree changes.

`git diff --staged`
 	
`git diff --cached`
	Also, to know the exact changes one is about to commit from the staging area of each file

`git diff <commit id 1> <commit id 2>`	To compare differences between two commit ids
`git diff <commit id 1> <commit id 2> <filename>`	This allows us to compare two commit ids for a particular filename
`Git difftool`	To know the exact changes made in working directory but not staged with difftool. Compares staging area with working tree changes. Please note that this doesn't work for docx files but gitdiff does.

`git difftool –cached`
 	
`git difftool --staged`
	Also, to know the exact changes one is about to commit from the staging area of each file with difftool

`git difftool <commit id 1> <commit id 2>`	To use a different tool for comparison like vimdiff, emerge, p4merge etc. This allows us to compare two commit ids
`git difftool <commit id 1> <commit id 2> <filename>`	This allows us to compare two commit ids for a particular filename
Tracking files	**Description**
git ls-files	To find out all the files being tracked by git
Git ls-files --other	To list all the untracked files in a directory
Viewing commit history	
`git log`	To see the history of commits in reverse chronological order of the current branch. This lists commit id, author name, email, date of commit and commit messages
`git log <branch name>`	To get commit history of a particular branch
`git log --all`	To see commit history of all the branches
`git log --pretty=oneline` or `git log --oneline`	Shows commit information (not file information) in one line. Shows a better readable formatting when many commits are present.
`git log --pretty=short`	Shows commit information (not file information) in short. Same as git log, but no commit date
`git log --pretty=full`	Shows committer and author information both in commit information (not file details)
`git log --pretty=fuller`	Committer and author information and the corresponding dates in commit information (not file details)
`git log --pretty=format:"%h %ar"`	To get the commit id, author, committer information, date, commit message in a specific format. Shows commit information (not file information)
`git log --relative-date`	The time relative to now when the commit was made instead of the absolute date (shows commit information and not file information)
`git log --stat`	In addition to commit information it displays file information like filename, no of insertions and deletions
`git log --shortstat`	In addition to commit information it displays file information with stats summary line for file changes
`git log --name-only`	In addition to commit information it displays file information with only file name in stat
`git log --name-status`	In addition to commit information it displays file information with filename and the action done on that file, like additions, modification or deletion.
`git log --graph`	To view branches and merges information in logs
`git log --decorate`	To see the current branch, i.e. to which branch the HEAD is pointing. Also, the current

	position in the branch is also shown (Not so useful as by default also HEAD is shown in latest git versions)
`git log -p -2`	Filter out commits. In addition to commit information and file information, it displays modification information to show the patch details of the last two commits
`git log -p -1 <commitid>`	Filter out commits. In addition to commit information and file information, it displays modification information to show patch details of a specific patch with commit id
`git log --author="<part of author name>"`	Filter out commits. To see commits by a specific author
`git log --committer="<part of committer name>"`	Filter out commits. To see commits by a specific committer
`git log -since="2 weeks"` ("2 years, 2 minutes, 2 hours or specify a specific date "2008-01-15" etc.)	Filter out commits. By time boundation.
`git log -until="2 weeks" ("2 years, 2 minutes, 2 hours or specify a specific date "2008-01-15" etc.)`	Filter out commits. By time boundation.
`git log |grep "message"`	Filter out commits. Greps message in commit messages and filters the commits
`git log <filename/directory name>`	Filter out commits. Show only those commits which are related to a specific file or directory
`git log --no-merges`	Filter out commits. Show only those commits which are not merge commits
`git show`	Useful to see the last patch applied. Often when 'git log –p -1' doesn't show the last patch information generated after a merge. This command can be used.
`git show <commitid>`	To show the details of a commit id.
`git log --oneline --graph --all`	This is equivalent of a non-existing command git history.
`git reset <commit id> --soft`	Moves the HEAD pointer to a particular commit id. The diff between the original HEAD position and the current position is saved in the staging area. Git log after this command will show timeline of commits done before this commit id and not the ones after.
`git reset <commit id> --mixed`	Moves the HEAD pointer to a particular commit id. The diff between the original HEAD position and the current position is saved in the working directory and staging area is blank. Git log after this command will show timeline of commits done before this commit id and not the ones after.
`git reset <commit id> --hard`	Moves the HEAD pointer to a particular commit id. The diff between the original HEAD position and the current position is not saved.

	Git log after this command will show timeline of commits done before this commit id and not the ones after.
`git reflog`	Stores all commit ids and actions taken in this repos in the entire history. So, this is a comprehensive history. Even deleted branches etc. can be tracked by this.
Git Aliases	**Description**
`git config --global alias.history "log --oneline --graph --all"`	Use git aliases to shorten a long command. In this example we create history alias for the log command to shorten it.
Undoing Things	
`git commit --amend`	To amend previous commit by additionally committing staging area contents. It can also help to modify the commit message. Here, the previous commit is deleted and replaced by a new commit.
Working with remotes	
`git remote`	To see the remote servers configured. If a repo is cloned then we should see atleast one remote repo. The default name of the repo server is 'origin' from which you cloned.
`git remote -v`	To see the url of the remote servers for both fetch and push
`git remote add <shortname> <url>`	To add a remote server. Shortname is usually 'origin'. But it can be changed too. Shortname is only the localrepo's name for the remote repo. The actual remote repo is a complete url.
`git remote rename <currentname> <newname>`	To rename a remote repository in the local repo records. The server of remote repo has no changes.
`git clone –o <name for remote repo>`	While cloning if we want a specific name of the remote repo, other than the default 'origin', it can be done like this
`git remote remove <remote repo name>` `git remote rm <remote repo name>`	To remove a remote repo. E.g. remote repot name is 'origin'.

`git push <remote repo> --delete <remote branch name>`
`or`
`git push <remote repo> :<remote branch name>`
	To delete a remote branch from a remote repo

`git fetch <remote repo name>`	Merges data from the remote repository to the local repository. No change is made to the working directory. They have to be merged manually. Only the remote-tracking branch is updated in the local repository and not the tracking branch. For multi-branch scenario, all remote-tracking branches are updated.

`git fetch -p`	This is helpful to prune those remote tracking branches for which the remote repo is deleted. Simple git fetch will not prune these branches.
`git fetch -all`	To fetch from all the remote repos instead of just one
`git pull <remote repo name>`	It does the job of fetch and additionally tries to merge the remote repo contents to the working directory. It is to be done only when all the changes are committed to the local branch. The remote-tracking and tracking branch, both are updated. For multi-branch scenario, the command can be configured to update all branches as well. But by default it updates the current local branch.
`git pull --all`	To update all local branches in a multi-branch scenario
`git checkout --ours -- path/to/file.txt` `git checkout --theirs -- path/to/file.txt`	For binary files when merge fails after fetch, one needs to decide which file one wants to keep from the two i.e. master or origin/master. These commands help to resolve that

`git push –u <remote> <branch>`
and
`git push <remote> <branch>`
	To push your changes upstream to the remote repository branch. E.g. of remote is 'origin' and branch is 'master'. It will work only if your local repo is up to date with the remote else the push request is rejected. Use fetch/pull to keep local repo updated. If the remote branch already doesn't exist, it will be created. The first push should use '-u' option to establish the tracking relationship between the local branch and the remote branch. After tracking branches are enabled, only git push command is enough to push the current branch (since git 2.0)

`git push --all`	To push all branches changes at once.

`git push <remote> refs/heads/<local branch name>:refs/heads/<remote branch name>`
or
`git push <remote> <local branch name>:<remote branch name>`
	It means push my local branch to remote repo under a different branch name

`git remote show <remote repo name>`	To show the relationship between local and remote repositories, i.e., if one pushes the changes from a local branch, the command helps identify the remote repo and the corresponding branch the change will go to. Similarly, vice-versa for git pull. Example 'remote repo name' is 'origin'. It also shows if the tracking branch and remote branches are

	up to date, behind or ahead as it establishes a connection with the server like fetch, pull and push commands
`git remote set-url origin <new url name>`	If the remote repository is updated, its remote references in the local repo also need to be updated. This is the method to update the url
Tagging	
`git tag`	Lists all the tags for a repository.
`git tag | grep "pattern*"`	List only those tags which have a pattern
`git tag –a <tagname> -m "<commit message>"`	Creates an annotated tag <tagname> (e.g. v1.0) with commit message and it also records taggers information
`git tag <tagname>`	Creates a tag of 'tagname' for the HEAD position. This is a lightweight tag. It is better to use annotated tags
`git tag <tagname> <commit id>`	To tag a particular commit id using a lightweight tag
`git tag -a <tagname> <commit id>`	Create an annotated tagname for a specific commit id.
`git tag –f <existing tagname> <commit id>`	Updating an existing tag to a new commit id on local repo.
`git show <tagname>`	To show detailed information about a particular tag e.g. the tagger, time of tag etc.
`git push <remote repo> <tagname>`	Since tags are not pushed by default. They need to be pushed explicitly. Remote repo example is 'origin'. This method pushes only one tag at a time.
`git push --force <remote repo> <tagname>`	To update an existing tag on git hub
`git push <remote repo> --tags`	Pushes all the tags (annotated and lightweight) to the remote server
`git tag -d <tagname>`	Delete a tag on a local repo

`git push <remote repo> -d <tagname>`
or
`git push <remote repo> :<tagname>`
	Delete a tag on remote repo

`git checkout <tagname>`	If you want to view the source code at a particular tag. It is not a good approach if you want to modify it too as the HEAD is in detached state
`git checkout -b <new branchname> <tagname>`	Create a new branch if you want to edit the files at a particular tagname
`git stash`	To stash the current state of a branch so that it can be used later again from the same state
`git stash list`	To list all the stashes

`git stash pop`	To pop the stash in the stack. It pops the latest entry to resume work from where it was last left
`git stash pop stash@{<stash number>}`	To pop a particular stash

Commands for branching	**Description**
`git branch <new branch name>`	To create a new local branch. The new branch created points to the same commit object where the current branch points to when the new branch was created. The current branch, however, is not switched
`git branch`	Shows all the branches in a local repo and also shows ur current branch marked with *
`git branch -a`	Shows all the local and remote branches
`git branch –d <branch name>`	To delete a branch. Works only when all changes are merged to another branch else it will fail
`git branch –D <branch name>`	To forcefully delete a branch even if its contents aren't merged yet.
`git branch --merged /--no-merged`	This is to find from the reference of the current branch. This finds all the other branches that are merged/not merged to it yet. This is useful to know for branch management, i.e. to delete them if they have been merged already
`git branch --merged/--no-merged <ref branch name>`	To see merged/no-merged branches for the branch 'ref branch name'
`git branch -vv`	To find out the local branches that are tracking branches, their status if they are up to date with remote-tracking branches, ahead or behind but not with the actual remote branches. No connection with the server is created to get this status. For getting an up to date status run 'git fetch' before.
`git log --decorate`	To see the current branch, i.e. to which branch the HEAD is pointing. Also, the current position in the branch is also shown (Not so useful as by default also HEAD is shown in latest git versions)
`git checkout <branch name>`	To switch from an existing branch to the new branch. Moves to the top commit of the branch. Also modifies the working directory accordingly. Always commit your changes in the current branch before you invoke git checkout of a new branch. If you are in the middle of something then stash your changes before switching. Branch switching won't work if you have uncommitted changes

`git checkout –b <new branch name>`	To create a new branch and switch to it

`git checkout –b <new local branch name> <remote repo>/<existing remote branch>`
Is the same as
`git checkout --track <remote repo>/<existing remote branch>`
same as
`git checkout <existing remote branch>`
	To create a new branch and switch to it.

`git commit –m "Comment.., close #<issue number>"`
`e.g.`
`git commit –m "Comment.., close #4"`
	To close an issue on GitHub with a commit id for easier tracking of Issues with their respective solutions. For issues already closed, a commit id can be associated by providing comment as in the example "Associating #3 with the issue"

`git checkout <commit id>`	To checkout a specific commit id. Leaves the HEAD in detached state. So, don't make changes here. If changes are necessary, create a branch instead.
`git merge <branch to be merged to another branch>`	Merges the content of the branch to be merged to the current branch. If "fast-forward" phrase is observed during merge then it implies the two branches weren't divergent and one was part of another's history. For merge conflict file identification, run 'git status'. The files with conflicts will contain symbols like "<<<" ">>>" and "===" delete these and resolve the conflict by editing this section and then do 'git add' to stage the changes for commit.
`git mergetool`	This is used to run a visual merging tool when a merge conflict is detected. This helps to do manual merge. The merge conflict files are automatically detected by the tool after a conflict is reported

`git checkout experiment`
`git rebase master`
`git checkout master`
`git merge experiment`
	Steps for rebasing a new development on branch 'experiment' onto 'master' branch which has diverged.

`git diff --check`	Check for white spaces while submitting a commit. Remove all unnecessary white spaces.

Other useful Information

Anything that is committed to the repo can always be recovered. Even if it was a commit that was deleted later. Anything lost that was not committed will be lost forever.

If a file is modified, staged and modified again, the commit will be done only for the changes which have been staged and not the latest ones in the working directory

Readme.md markdown file is rendered by Github Automatically

Fullscreen mode of editing a source = Zen mode in Github

Use Topic branches for development always.

Rebase commits only for your local repo. Don't do it for remote repos.

Avoid creating topic branch of a topic branch and then do a rebase. It is doable but adds unnecessary level of complexity.

Before push do a git fetch and merge always (or git pull)

Use Gists feature to share a code snippet with anyone. It acts like a traditional git repository.