Set up Git

git config --global user.name "John Doe" ⇒ *Set user name*
git config --global user.email johndoe@example.com ⇒ *Set user address*
git config --global core.editor emacs ⇒ *Set editor*

Your default branch name:

By default Git will create a branch named master when you create a new repository with git init.
But From Git version 2.28, you can set a different name for the initial branch: git config --global init.defaultBranch main

Checking Your Settings:

git config --list
You can also check what Git thinks a specific key's value is by typing git config <key>, example: git config user.name

I .Git Basic:

Getting a Git Repository:

- Initializing a Repository in an Existing Directory:
⇒ *go to the directory: cd /home/user/my_project and type: $ git init*
- Cloning an Existing Repository:
⇒ *$ git clone <url>*

Checking the Status of Your Files:

The main tool you use to determine which files are in which state is the git status command: $ git status

Tracking New Files:

In order to begin tracking a new file, you use the command: $ git add.
To begin tracking the README file, you can run this: $ git add README Then run: $ git status ⇒ *new file: README*

Staging Modified Files:

Let's change a file that was already tracked. file.c then run: $ git status.
⇒ *Changes not staged for commit*
modified: file.c
To stage it, you run: $ git add .

Ignoring Files:

.gitignore file: ending in ".o" or ".a" — object and archive files that may be the product of building your code. or ignore TODO file, pdf file ...

git diff:

to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit?
$ git diff: shows you the exact lines added and removed — the patch, as it were.
To see what you've changed but not yet staged, type $ git diff with no other arguments
If you want to see what you've staged that will go into your next commit, you can use $ git diff --staged

Committing Your Changes:

Now that your staging area is set up the way you want it, you can commit your changes.
Remember that anything that is still unstaged — any files you have created or modified that you haven't run git add on since you edited them — won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran git status, you saw that everything was staged, so you're ready to commit your changes:
$ git commit.
⇒ *Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.*

Skipping the Staging Area:

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part: $ git commit -a

Removing Files:

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit: $ rm file
⇒ *it shows up under the "Changes not staged for commit" (that is, unstaged) area. deleted:file*
if you run $ git rm, it stages the file's removal
The next time you commit, the file will be gone and no longer tracked. If you modified the file or had already added it to the staging area, you must force the removal with the -f option.

Moving Files:

mv command: rename a file in Git: $ git mv file_from file_to

Viewing the Commit History:

$ git log ⇒ lists the commits

$ git log -p -2: which shows the difference (the patch output) introduced in each commit.

$ git log --stat: prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed.

Undoing Things:

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option: $ git commit --amend

⇒ As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

$ git commit -m 'Initial commit'

$ git add forgotten_file

$ git commit --amend

Unstaging a Staged File:

$ git reset HEAD <file> or $ git restore --staged <file>

Un-modifying a Modified File

How can you easily unmodify file — revert it back to what it looked like when you last committed:

$ git checkout -- <file>

!!! It's important to understand that git checkout -- <file> is a dangerous command.Any local changes you made to that file are gone

- Git just replaced that file with the most recently-committed version.

II Working with Remotes:

Showing Your Remotes:

To see which remote servers you have configured, you can run: $ git remote

*⇒ you should at least see **origin** — that is the default name Git*

Adding Remote Repositories:

$ git remote add <shortname> <url>

⇒ then run: $ git remote, you will see origin & shortname.

if you want to fetch all the information that <shortname> has: $ git fetch <shortname>

Result:

** [new branch] master -> pb/master*

** [new branch] ticgit -> pb/ticgit*

⇒ Paul's master branch is now accessible locally as pb/master — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it.

Fetching and Pulling from Your Remotes:

$ git fetch <remote>:

⇒ The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet.

*If you clone a repository, the command automatically adds that remote repository under the name **"origin".***

$ git fetch origin: fetches any new work that has been pushed to that server since you cloned (or last fetched from) it.

It's important to note that the git fetch command only downloads the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on.

Pushing to Your Remotes:

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple:

$ git push <remote> <branch>

If you want to push your master branch to your origin server: $ git push origin master

Inspecting a Remote:

If you want to see more information about a particular remote, you can use: $ git remote show <remote> .

if you run this command with a particular shortname, such as origin: $ git remote show origin

Renaming and Removing Remotes:

You can run git remote rename to change a remote's shortname: $ git remote rename pb paul (change pb to paul)

If you want to remove a remote for some reason, you can either use: $ git remote remove or $ git remote rm.

Tagging:

Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

Listing Your Tags: $ git tag

Annotated Tags:

Creating an annotated tag in Git: $ git tag -a v1.4 -m "my version 1.4"

Now, suppose you forgot to tag the project at v1.2, which was at the "Update rakefile" commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

$ git tag -a v1.2 9fceb02 (commit checksum)

Sharing Tags:

By default, the git push command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run:

$ git push origin <tagname>

To push all tags at once: $ git push origin --tags

Deleting Tags:

$ git tag -d <tagname>

==> Note that this does not remove the tag from any remote servers. There are two common variations for deleting a tag from a remote server:

1. The first variation is: $ git push <remote> :refs/tags/<tagname>, example: $ git push origin :refs/tags/v1.4-lw

2. The second (and more intuitive) way to delete a remote tag is with: $ git push origin --delete <tagname>

Checking out Tags:

$ git checkout <tagname> (ex: v2.0.0)

III. Git Branching

The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

Creating a New Branch:

$ git branch <name of branch>

⇒ This creates a new pointer to the same commit you're currently on.

How does Git know what branch you're currently on? It keeps a special pointer called HEAD

*$ git log: how you where the branch pointers are pointing. **(HEAD -> master, testing)***

Switching Branches

To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

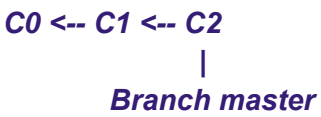
$ git checkout testing

This moves HEAD to point to the testing branch.

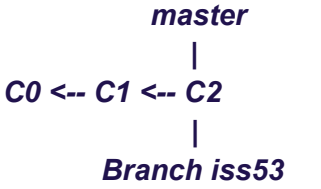
⇒ Create a new branch and want to switch to that new branch at the same time: $ git checkout -b <new branch name>

Basic Branching:

First, let's say you're working on your project and have a couple of commits already on the master branch:



You've decided that you're going to work on issue #53: $ git checkout -b iss53

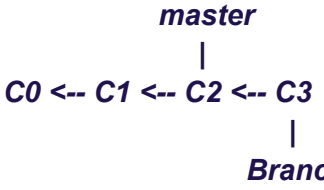


⇒ your HEAD is pointing to iss53

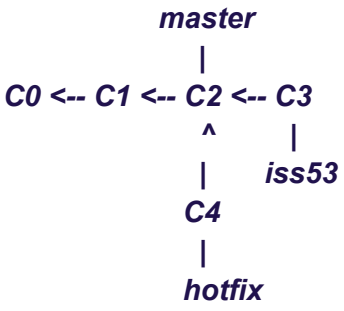
Suppose, we did change like:

$ vim index.html

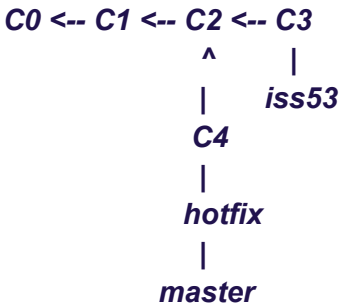
$ git commit -a -m 'Create new footer [issue 53]'



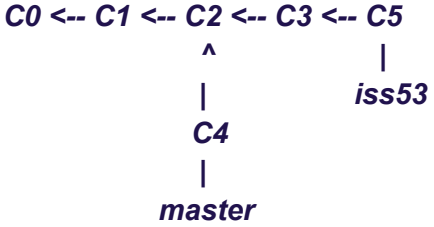
$ git checkout master:
To remember: when you switch branches, Git resets your working directory to look like it did the last time you committed to that branch.
$ git checkout -b hotfix :Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'



You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production.
$ git checkout master
$ git merge hotfix
Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.



After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted(iss53). However, first you'll delete the hotfix branch, because you no longer need it — the master branch points at the same place.
$ git branch -d hotfix
Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.
$ git checkout iss53 :Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'



Basic Merging:
Suppose you've decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you'll merge your iss53 branch into master:
$ git checkout master ⇒ Switched to branch 'master'
$ git merge iss53
you have no further need for the iss53 branch. You can close the issue in your issue-tracking system, and delete the branch:
$ git branch -d iss53
Basic Merge Conflicts:
If your fix for issue #53 modified the same part of a file as the hotfix branch, you'll get a merge conflict.
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run: $ git status
Anything that has merge conflicts and hasn't been resolved is listed as unmerged.
After you've resolved each of these sections in each conflicted file, run: $ git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.
⇒ If you want to use a graphical tool to resolve these issues, you can run: $ git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts.
⇒ After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you.
You can run: $ git status again to verify that all conflicts have been resolved.then type: $ git commit to finalize the merge commit

Branch Management:

The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches: $ git branch

```
iss53
* master
testing
```

*⇒ Notice the * character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to).*
To see the last commit on each branch, you can run: $ git branch -v.
To see which branches are already merged into the branch you're on, you can run: $ git branch --merged
*Branches on this list without the * in front of them are generally fine to delete with: $ git branch -d; you've already incorporated their work into *master.*
To see all the branches that contain work you haven't yet merged in, you can run: $ git branch --no -merged:
*Because it contains work that isn't merged in yet, trying to delete it with **git branch -d will fail:***
If you are sure you want to delete it, you can force it with -D , run '$ git branch -D testing'.

Changing a branch name:

⇒ Do not rename branches that are still in use by other collaborators.
Rename branch: $ git branch --move old-name new-name
But this change is only local for now. To let others see the corrected branch on the remote, push it:
$ git push --set-upstream origin new-name
Notice that you're on the branch corrected-branch-name. The corrected branch is available on the remote. However the bad branch is also still present on the remote. You can delete the bad branch from the remote:
$ git push origin --delete new-name

Changing the master branch name:

⇒ Before you do this, make sure you consult with your collaborators.
Rename your local master branch into main with the following command: $ git branch --move master main
To let others see the new main branch, you need to push it to the remote. This makes the renamed branch available on the remote.
$ git push --set-upstream origin main
The main branch is also available on the remote. But the remote still has a master branch. Other collaborators will continue to use the master branch as the base of their work.
If you are certain the main branch performs just as the master branch, you can delete the master branch in remote:
$ git push origin --delete master

Remote Branches

You can get a full list of remote references with: $ git ls-remote <remote>, or $ git remote show <remote>
*Remote-tracking branch names take the form <remote>/<branch>. ex: **origin/master***
*To synchronize your work with a given remote, you run: $ git fetch <remote> (in our case, **$ git fetch origin**). This command looks up which server **"origin"** is (ex: git.ourcompany.com), fetches any data from it that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.*

***Pushing**:*

*If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run: **$ git push <remote> <branch>**: ex: $ git push origin serverfix.*
To merge this work into your current working branch, you can run: $ git merge origin/serverfix

Tracking Branches:

When you clone a repository, it generally automatically creates a master branch that tracks origin/master.
However, you can set up other tracking branches:

$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin. Switched to a new branch 'serverfix'
To set up a local branch with a different name than the remote branch: $ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin. Switched to a new branch 'sf'
⇒ Now, your local branch sf will automatically pull from origin/serverfix.
If you already have a local branch and want to set it to a remote branch: $ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
If you want to see what tracking branches you have set up: $ git branch -vv
This will list out your local branches with more information including what each branch is tracking.
Pulling:
$ git pull = $ git fetch + $ git merge.
It will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.
Deleting Remote Branches:
$ git push origin --delete serverfix
***Rebasing**:*
In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.
With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch.
For this example, you would check out the experiment branch, and then rebase it onto the master branch as follows:
$ git checkout experiment
$ git rebase master
At this point, you can go back to the master branch and do a fast-forward merge.
$ git checkout master
$ git merge experiment

$ git rebase --onto master server client: use --onto when we have multiple divergion
$ git checkout master
$ git merge client
finally:
$ git rebase master server
$ git checkout master
$ git merge server
⇒ You can remove the client and server:
$ git branch -d client
$ git branch -d server
!!!!! Do not rebase commits that exist outside your repository and that people may have based work on.!!!!

Private Small Team
Two developers start to work together with a shared repository: John and Jessica
 # John's Machine
$ git clone john@githost:simplegit.git
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remove invalid default value'

 # Jessica's Machine
$ git clone jessica@githost:simplegit.git
$ cd simplegit/
$ vim TODO
$ git commit -am 'Add reset task'
⇒ Now, Jessica pushes her work to the server
 # Jessica's Machine
$ git push origin master

⇒ John try to push the change to the same server:

 # John's Machine
$ git push origin master
⇒ John's push fails because of Jessica's earlier push of her changes.
So, you must first merge the commits locally. In other words, John must first fetch Jessica's upstream changes and merge them into his local repository before he will be allowed to push.
 # John's Machine
$ git fetch origin
Now John can merge Jessica's work that he fetched into his own local work:
 # John's Machine
$ git merge origin/master
Now John test the new code of jessica and if everything seems fine, he can push:
 # John's Machine
$ git push origin master

- Jessica has created a new topic branch called issue54, and made three commits to that branch.
⇒ so she can fetch all new content from the server
 # Jessica's Machine
$ git fetch origin
Now Jessica wants to know what part of John's fetched work she has to merge into her work so that she can push:
$ git log --no-merges issue54..origin/master
*⇒ The issue54..origin/master syntax is a log filter that asks Git to display only those commits that are on the latter branch (**in this case origin/master**) that are not on the first branch (**in this case issue54**)*
Now, Jessica can merge her topic work into her master branch:
$ git checkout master
$ git merge issue54
*⇒ Jessica now completes the local merging process by merging John's earlier fetched work that is sitting in the **origin/master** branch:*
$ git merge origin/master
⇒ Now Jessica able to successfully push:
$ git push origin master

Private Managed Team
John and Jessica: Feature A
Josie and jessica: Feature B
⇒ Let's follow Jessica's workflow as she works on her two features.Assuming she already has her repository cloned she decides to work on featureA first.
 # Jessica's Machine
$ git checkout -b featureA :Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'Add limit to log function'
*⇒ At this point, she needs to share her work with John, so she pushes her featureA branch commits up to the server. Jessica doesn't have push access to the master branch — **only the integrators do** — so she has to push to another branch in order to collaborate with John:*
$ git push -u origin featureA (feature A is new branch)
Jessica emails John to tell him that she's pushed some work into a branch named featureA and he can look at it now. While she waits for feedback from John, Jessica decides to start working on featureB with Josie.
 # Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Now, Jessica makes a couple of commits on the featureB branch:
$ vim lib/simplegit.rb
$ git commit -am 'Make ls-tree function recursive'
$ vim lib/simplegit.rb
$ git commit -am 'Add ls-files'
*She's ready to push her work, but gets an email from Josie that a branch with some initial "featureB" work on it was already pushed to the server as the **featureBee** branch. Jessica needs to merge those changes with her own before she can push her work to the server. Jessica first fetches Josie's changes with:*

$ git fetch origin
Assuming Jessica is still on her checked-out featureB branch, she can now merge Josie's work into that branch with
$ git merge origin/featureBee
Now Jessica can push:
$ git push -u origin featureB:featureBee
Suddenly, Jessica gets email from John, who tells her he's pushed some changes to the featureA branch
$ git fetch origin
Jessica can display the log of John's new work by comparing the content of the newly-fetched featureA branch with her local copy of the same branch:
$ git log featureA..origin/featureA
⇒ If Jessica likes what she sees, she can merge John's new work into her local featureA branch with:
$ git checkout featureA : Switched to branch 'featureA'
$ git merge origin/featureA
Finally, Jessica might want to make a couple minor changes to all that merged content, commit them to her local featureA branch, and push the end result back to the server:
$ git commit -am 'Add small tweak to merged content'
$ git push
⇒ At some point, Jessica, Josie, and John inform the integrators that the featureA and featureBee branches on the server are ready for integration into the mainline. After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit.

Forked Public Project
Contributing to public projects is a bit different. Because you don't have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way.
⇒ First, you'll probably want to clone the main repository, create a topic branch for the patch
$ git clone <url>
$ cd project
$ git checkout -b featureA
$ git commit
When your branch work is finished and you're ready to contribute it back to the maintainers, go to the original project page and click the "Fork" button, creating your own writable fork of the project. You then need to add this repository URL as a new remote of your local repository
$ git remote add myfork <url>
You then need to push your new work to this repository.
$ git push -u myfork featureA
Once your work has been pushed to your fork of the repository, you need to notify the maintainers of the original project that you have work you'd like them to merge. This is often called a pull request, or you can run the git request-pull command and email the subsequent output to the project maintainer manually.
⇒
The git request-pull command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and produces a summary of all the changes you're asking to be pulled. For instance, if Jessica wants to send John a pull request, and she's done two commits on the topic branch she just pushed, she can run this:
$ git request-pull origin/master myfork
The output can be sent to the maintainer — it tells them where the work was branched from, summarizes the commits, and identifies from where the new work is to be pulled.
For example, if you want to submit a second topic of work to the project, don't continue working on the topic branch you just pushed up — start over from the main repository's master branch:
$ git checkout -b featureB origin/master
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
$ git fetch origin
Let's say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of origin/master, resolve the conflicts for the maintainer, and then resubmit your changes:

$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
If you want to change implementation detail, You start a new branch based off the current ***origin/master*** *branch, squash the featureB changes there, resolve any conflicts, make the implementation change, and then push that as a new branch:*
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
$ git commit
$ git push myfork featureBv2

How to Push an Existing Project to GitHub
Step 1: ***Create a new GitHub Repo***

Step 2: ***Initialize Git in the project folder:***
$ git init

Step 3: ***Add the files to Git index:***
$ git add -A
-A: *means "include all".*

Step 4: ***Commit Added Files:***
$ git commit -m 'Added my project'

Step 5: ***Add new remote origin (in this case, GitHub)***
$ git remote add origin git@github.com:sammy/my-new-project.git
⇒ *In git, a "remote" refers to a remote version of the same repository, which is typically on a server somewhere (in this case GitHub.) "origin" is the default name git gives to a remote server (you can have multiple remotes).*

Step 6: ***Push to GitHub***
$ git push -u -f origin master
⇒ *The -f flag stands for force. This will automatically overwrite everything in the remote directory.*
⇒ *The -u flag sets the remote origin as the default.*