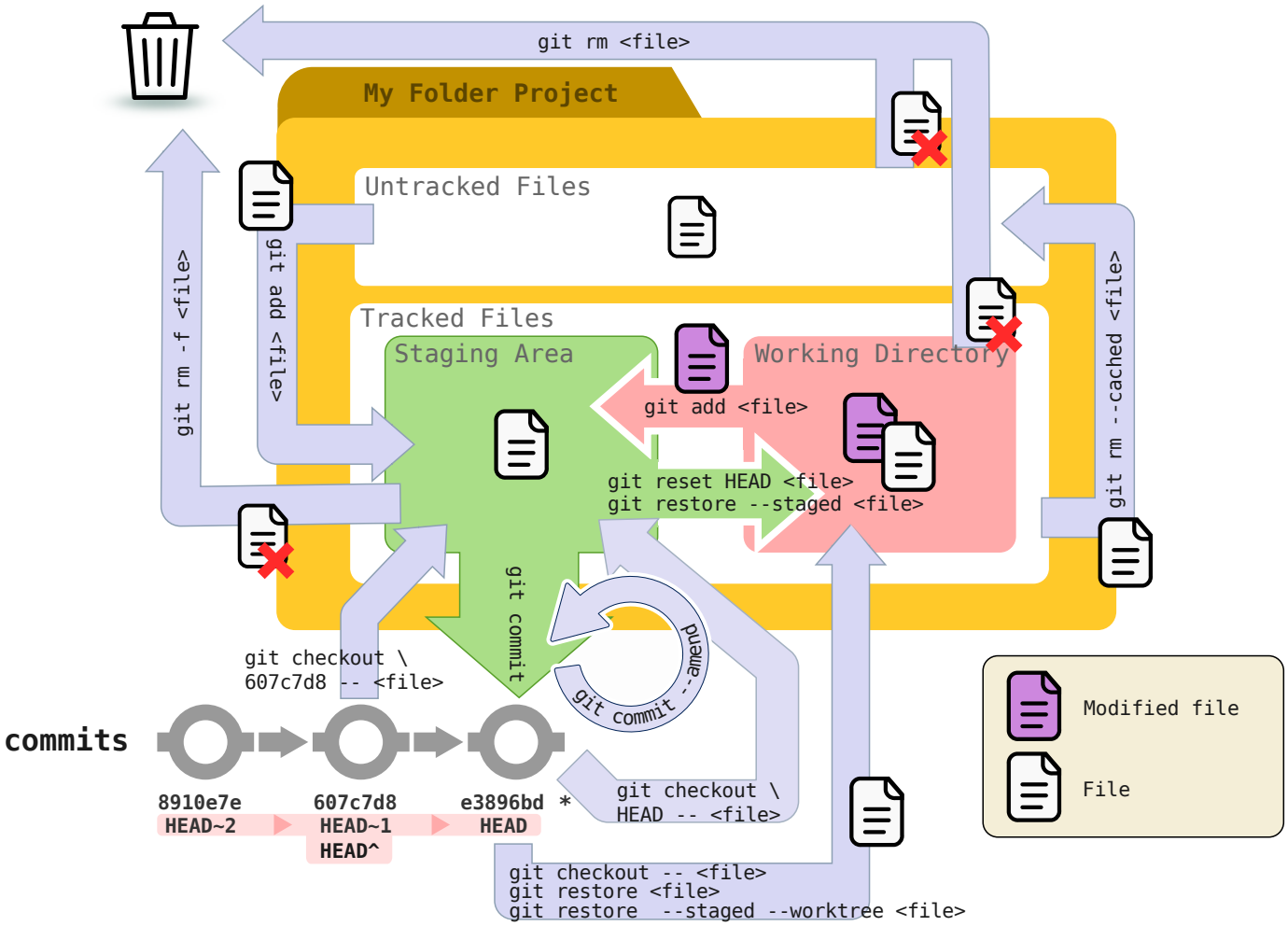


Config

Setup

```
git config --global user.name "username"
git config --global user.email "somemail"
ssh-keygen -t rsa -C "somemail"
git config --global core.editor "vim"
git config --global init.defaultBranch main
```

Show config

```
git config --list
```

Start a working area

Start a project

```
git init
```

Download a remote repo

```
git clone <url>
```

Show things

Files in git can be tracked, untracked, staged, or unstaged. The command git status allows you to check the status of the files.

```
git status
```

Git offers an option to get a shortened status with **git status --short** or with **-s** option.

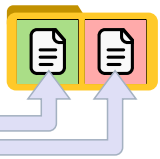
```
git status -s
```

Git status show the branch, changes to be commited, changes on working directory and untracked files as shown bellow

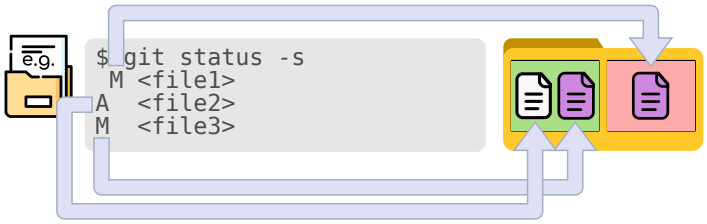

```
$ git status
On branch main
Changes to be committed:
...
Changes not staged for commit:
...
Untracked files:
...
```

The status is displayed in two columns, the left column indicates the staged state and the right column indicates the unstaged state or working directory.


```
$ git status -s
XX <file>
```

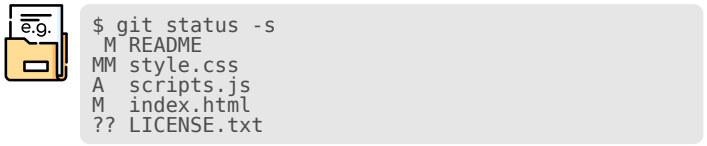


Staged files have an A and modified files have an M.



Files that are not tracked have a ?? at his side.

In the next example, the **README** file was modified in the working directory but is not ready, while **index.html** was modified and prepared. The **style.css** was modified, prepared, and modified again so there are prepared and unprepared changes. **scripts.js** is prepared



Show the changes on files of the working directory since the last commit
```
git diff
```

Show the changes on staged files
```
git diff --staged
```

Revert the changes made in HEAD
```
git revert HEAD
```

Revert changes made since a specific commit
```
git revert --no-commit HEAD   (Until arrive to the commit)
git revert --continue         (to confirm)
```

Reference to the parent of HEAD
```
HEAD^
```

Reference to the <#> commit before HEAD
```
HEAD~<#>
```

Show commit history
```
git log
```

Show changes made in each commit
```
git log -p
```

Show last 2 commits
```
git log -2
```

Show statistics of each commit
```
git log --stat
```

Show formated log output
```
git log --pretty=format:"... options ..."
```

Some format option:
```
 %H: Commit hash
 %h: Shortened commit hash
%ae: Author's email address
%cn: Confirmer name
%ce: Confirmer email address
%cd: Confirmation date
%cr: Commit date, relative
 %s: Subject
```


```
$ git log --pretty=format:"%h - %an, %ad: %s" \
--author="dasesu" --date=format:'%Y-%m-%d %H:%M'
6847082 - dasesu, 2024-06-29 11:15: second com..
0d14b0f - dasesu, 2024-06-28 18:57: first comm..
6b67385 - dasesu, 2024-06-28 13:32: Starting p..
```

Options like **--since** and **--until** allows you to filter by date ranges
```
git log --since=<period of time or date>
```


```
$ git log --pretty="%h - %s" --author=dasesu \
--since="2008-10-01" --before="2008-11-01"
```

Make changes

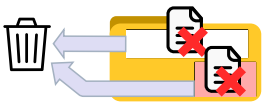
Add a file to stage
```
git add <file>
```

Stage all files
```
git add .
```

Stage the files by hunks
```
git add --patch
```

Some optiones:
```
y: stage this hunk
n: do not stage this hunk
q: quit; do not stage this hunk or any of the
remaining ones
a: stage this hunk and all later hunks in the
file
d: do not stage this hunk or any of the later
hunks in the file
s: split the current hunk into smaller hunks
e: manually edit the current hunk
?: print help
```

Remove a file
```
git rm <file>
```

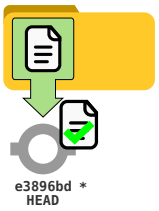


Remove a staged file
```
git rm -f <file>
```

Untrack files
```
git rm --cached <file>
```

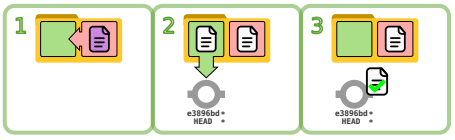
Change the name of a file
```
git mv <old_name> <new_name>
```

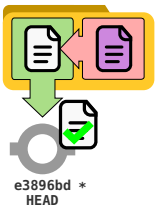
Save all staged files into a new commit
```
git commit
```



Save all staged files into a new commit and specify the commit message.
```
git commit -m "commit message"
```

Stage all tracked files and commit
```
git commit -am "commit message"
```

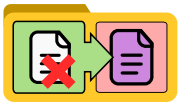




Redo the last commit, if changes were made they will be committed
```
git commit --amend
```

e.g.
```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

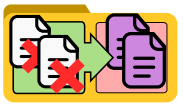
Move a file from staging area to working directory
```
git reset HEAD <file>
```



By default, git restore doesn't has effect over staged files, for restore them we need to add the **--staged** parameter
```
git restore --staged <file>
```

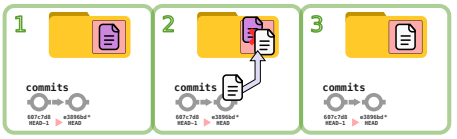
If instead of HEAD, we pass the identifier of the commit, for example d7dc9f4, to the git reset command, it will revert the project to the state of commit d7dc9f4, removing subsequent commits and setting d7dc9f4 as the new HEAD.

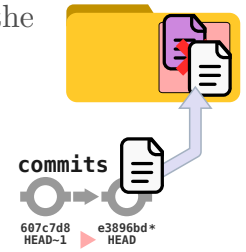
Move all files from staging area to Working Directory
```
git reset HEAD
```
Or
```
git restore --staged .
```



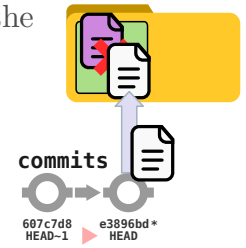
Restore all files to their state at the last commit.
```
git checkout -- .
```
Or
```
git restore .
```

Restore a file to their state at the last commit.
```
git checkout -- <file>
```
Or
```
git restore <file>
```

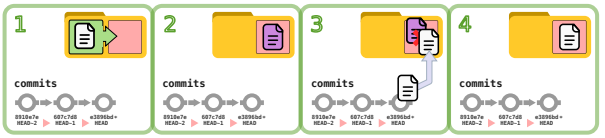


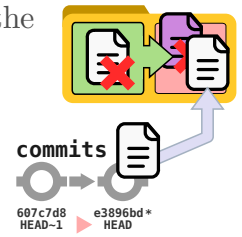


Restore a file to the state of the last commit into staging area
```
git checkout HEAD -- <file>
```

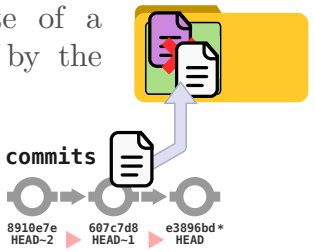


Restore a file to the state of the last commit overwriting stage
```
git restore  --staged \
--worktree <file>
```
Same as:
```
git reset HEAD <file>
git checkout -- <file>
```

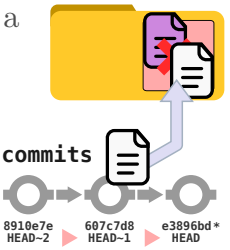




Restore a file to the state of a previous commit identified by the commit ID.
```
git checkout 607c7d8 \
 -- <file>
```
Same for all files
```
git checkout 607c7d8 \
-- .
```



Restore a file to the state of a previous commit
```
git restore --source=607c7d8 \
  <file>
```



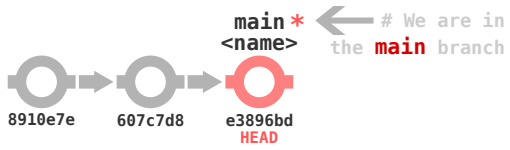
Working with Branches

List all local branches.
```
git branch
```

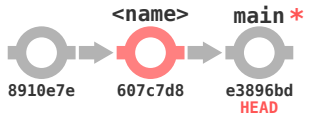
List all remote branches.
```
git branch -r
```

List all branches.
```
git branch --all
```

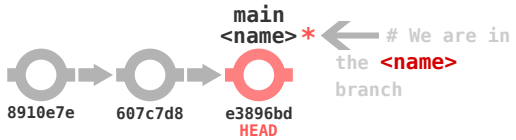
Create a new branch
```
git branch <name>
```



Create a new branch from a commit point
```
git branch <name> <commit-ID>
```

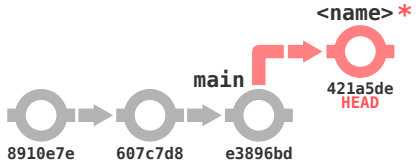


Switch to a branch and update working directory
```
git checkout <branch>
```
Or
```
git switch <branch>
```



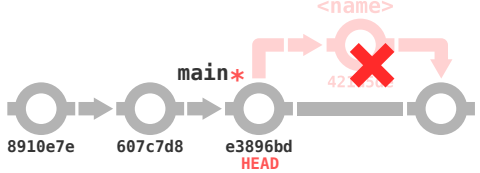
Create a new branch and switch to it
```
git checkout -b <branch>
```
Or
```
git switch -c <branch>
```

Add a new commit to the current branch
```
git commit
```



Change the branch name
```
git branch -m <old-name> <new-name>
```

Delete a branch
```
git branch -d <name>
```

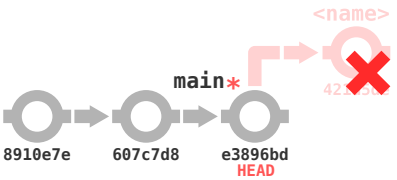


You must be in a different branch than the one you want to delete, so you need to switch to another branch before delete a branch as shown below

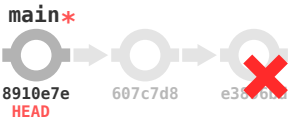

```
$ git branch
  main
* <name>
$ git switch main
Switched to branch 'main'
$ git branch -d <name>
```

For delete branches that have not been fully merged you must indicate the delete option with **-D** instead of **-d**

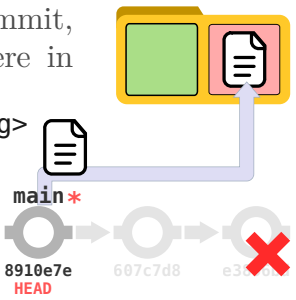
Delete branch (even if not merged)
```
git branch -D <name>
```

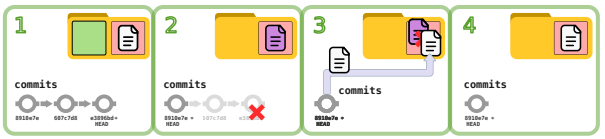


Reset the branch from a certain commit
```
git reset <commit-id or tag>
```
Or
```
git reset --soft <commit-id>
```



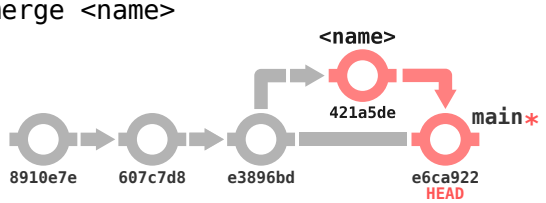
Reset the branch from a commit, updating the files as they were in the specified commit
```
git reset <commit-id or tag>
git checkout -- <file>
```
Or just
```
git reset --hard <commit>
```





Merging

Merge two branches
`git merge <name>`




```
Merge branch <name> into branch main.
```

To merge something from branch B to branch A, you must be on branch A and do git merge to branch B, as shown below

e.g.

```
$ git branch
  main
* <name>
$ git switch main
Switched to branch 'main'
$ git merge <name>
```

Merge and squash all commits into a new commit
`git branch --squash <name>`

Show the information of the last commit in each branch
`git branch -v`

e.g.

```
$ git branch -v
  <name> 421a5de fix issue
* main e6ca922 Merge branch '<name>'
  testing 782fd34 add file readme.md
```

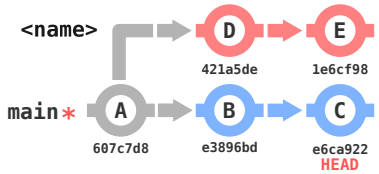
Show merged branch only
`git branch --merged`

Show non merged branch only
`git branch --no-merged`

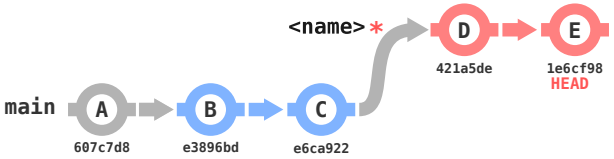
Rebasing

Git rebase allows you to integrate changes from one branch into another replaying the commit history

For example, giving the next situation

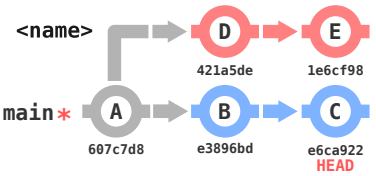


`git rebase main` would produce the next result



To apply rebase you must move into the branch you want to apply, and call rebase over the branch where you want the changes to be applied

For example, giving the next situation

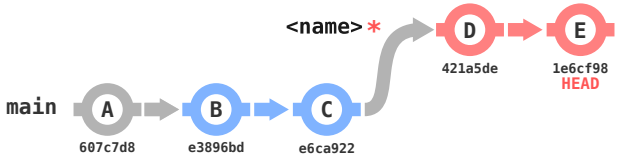


The procedure would be like is shown in the following commands

e.g.

```
$ git branch
* main
  <name>
$ git switch <name>
Switched to branch '<name>'
$ git rebase main
Successfully rebased and updated
```

Getting the next result



Interactive rebase allow you to customize the commits order
`git rebase -i <from-commit>`

You need to specify the commit from which start to apply the rebase, the easy way to do this is using the HEAD~<NUMBER> notation, where <NUMBER> refers to the number of steps before HEAD

For example the command
`git rebase -i HEAD~4`
This open a file showing somthing like this:

e.g.

```
pick e3896bd B from main
pick e6ca922 C from main
pick 421a5de D from <name>
pick 1e6cf98 E from <name>
# Rebase 421a5de..1e6cf98 onto 42...
...
```

Git read this file and follow the next **rules**

1) These lines are executed from top to bottom, and they can be re-ordered for change the commits order

2) Remove a line WILL LOST THAT COMMIT

3) Remove everything will abort the rebase.

4) You can modify the commits history by changing the pick option for another option.

Some rebase options are:

```
pick = use commit
reword = edit the commit message
edit = use commit, but stop for amending
squash = use commit, but meld into previous commit
```

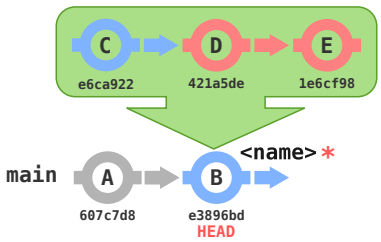
If you want to flatten the commit history. You can edit the file adding the word squash in front of each of the commits that you want to eliminate (flatten)

For example

e.g.

```
pick e3896bd B from main
squash e6ca922 C from main
squash 421a5de D from <name>
squash 1e6cf98 E from <name>
# Rebase 421a5de..1e6cf98 onto 42...
...
```

The result squash the selected commits into the one marked as pick, as describe the below image

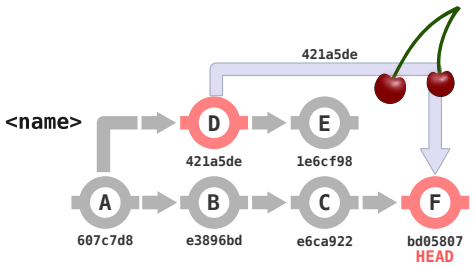


Cherry Picking

Allows you to select individual commits to be integrated

Cherry Pick a commit

```
git cherry-pick <commit-ID>
```



To cherry pick a commit from branch B to branch A, you must be on branch A and call the cherry-pick command passing the <commit-id> of the wanted commit

e.g.

```
$ git branch
*  main
  <name>
$ git switch <name>
Switched to branch '<name>'
$ git log --pretty=format:%h
1e6cf98
421a5de
$ git switch main
$ git cherry-pick 421a5de
```

Reflog

Is a protocol of HEAD pointer movements. Allows you to inspect the history of pointer movements and revert the deleted commits by reseting to the wanted state

```
git reflog
```

In the next example you can see a project with three commits **a60db64**, **164e0f4** and **48d08b8**. After hard reset over the commit **164e0f4** the commit **a60db64** will be lose. You can use reflog to know the the id of the deleted commit for recover the project to the previous state.

e.g.

```
$ git log --pretty=format:"%h %an %s"
a60db64 dasesu adding some basic css settings
164e0f4 dasesu Adding asset folder
48d08b8 dasesu initial project state
```


```
$ git reset --hard 164e0f4
HEAD is now at 164e0f4 Adding asset folder

$ git log --pretty=format:"%h %an %s"
164e0f4 dasesu Adding asset folder
48d08b8 dasesu initial project state
```


```
$ git reflog
164e0f4 (HEAD -> main, featureA) HEAD@{3}: res
a60db64 HEAD@{4}: commit: adding some basic cs
164e0f4 (HEAD -> main, featureA) HEAD@{5}: mer
48d08b8 HEAD@{6}: checkout: moving from featur
164e0f4 (HEAD -> main, featureA) HEAD@{7}: com
48d08b8 HEAD@{8}: checkout: moving from main t
48d08b8 HEAD@{9}: commit (initial): initial pr

$ git reset --hard a60db64
HEAD is now at a60db64 adding some basic css s

$ git log --pretty=format:"%h %an %s"
a60db64 dasesu adding some basic css settings
164e0f4 dasesu Adding asset folder
48d08b8 dasesu initial project state
```

Tags

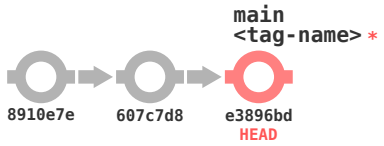
Git has the ability to tag specific points of the repository's history as being important

List the tags
```
git tag
```
Or
```
git tag -l
```

Filter the result with glob expressions
```
git tag -l 'v1.8.5*'
```

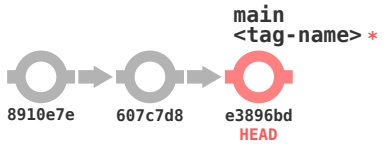
Git supports two types of tags: **lightweight** and **annotated**
A **lightweight** tag is very much like a branch that doesn't change
Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG)

Create an annotated tag in Git
```
git tag -a <tag-name> -m "tag message"
```

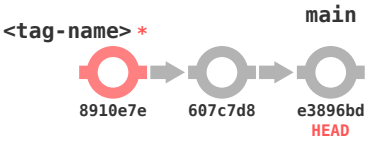


e.g.
```
$ git tag -a v1.0 -m "my awesome project v 1.0"
$ git tag
v1.0
```

Create a lightweight tag in Git
```
git tag <tag-name>
```



You can also tag commits after you've moved past them.
```
git tag <tag-name> <commit-ID>
```



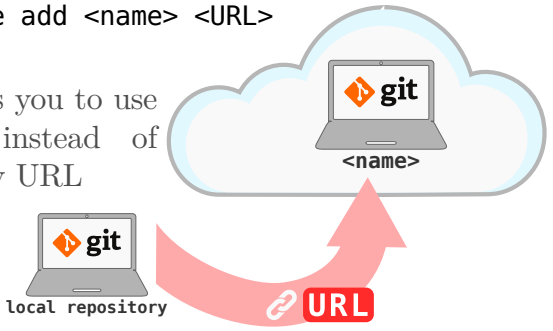
For share tags you will have to explicitly push tags to a shared server after you have created them
```
git push origin  <tag-name>
```

For share a lot of tags at once, you can also use the --tags option to the git push command
```
git push origin --tags
```

Working Remotely

Add a remote repository
```
git remote add <name> <URL>
```

This allows you to use <name> instead of the entirely URL



Remote repositories are like branches, when a remote repository is added, git create a branch associated to that remote repo. To list those branches you can use **-r** option or **--all**
```
git branch -r
```

Show all remote connections
```
git remote
```

Show all remote repository's URL
```
git remote -v
```

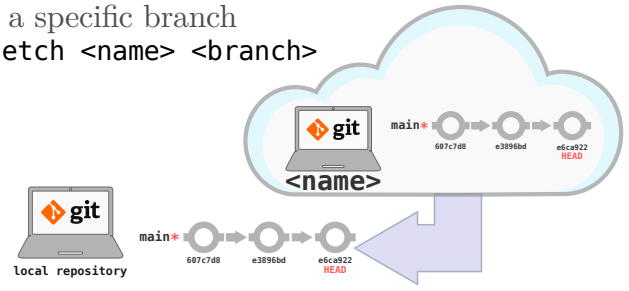
Remove a remote repository
```
git remote rm <name>
```

Rename a remote repository
```
git remote rename <name> <new-name>
```

Shows the status of the remote repository with respect to the local repository
```
git remote show <name>
```

e.g.
```
$ git remote show <name>
remote <name>
Fetch URL: some url
Push  URL: some url
HEAD branch: main
Remote branches:
  somebranch new (next fetch will store in ...
  main       tracked
Local refs configured for 'git push':
  somebranch pushes to somebranch (up to date)
  main       pushes to main       (up to date)
```

Fetch a specific branch
```
git fetch <name> <branch>
```



The fetch command download the data that it's in the remote repo but not in the local repo. Since we are often working on the same files, we also have to merge them using merge or rebase.

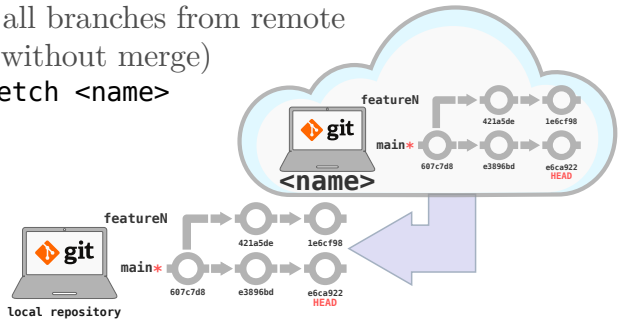
```
git fetch <name> <branch>
git merge <name>/<branch>
```

Fetchs the remote repo's copy of the specified branch, then merge

```
git pull <name> <branch>
```

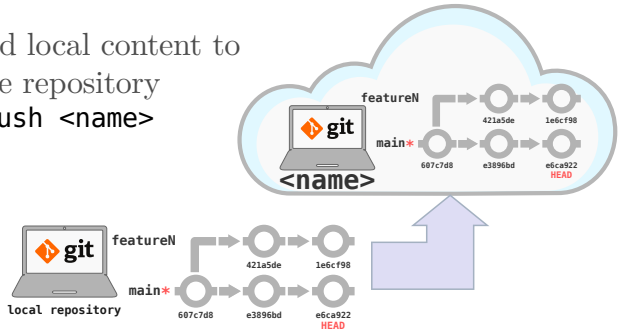
Fetch all branches from remote repo (without merge)

```
git fetch <name>
```



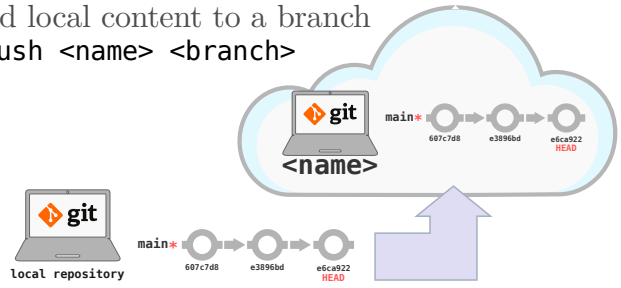
Upload local content to remote repository

```
git push <name>
```



Upload local content to a branch

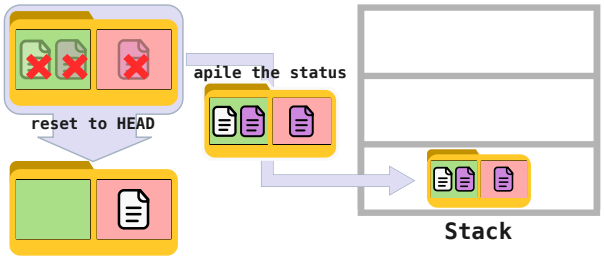
```
git push <name> <branch>
```



Stashing

Store modified & stage changes. To include untracked files, add -u flag. For untracked and ignored files add -a f

```
git stash
```



As above, but add a comment

```
git stash save "some description"
```

Partial Stash, Stash just a single file, a collection of files, or individual changes from within files

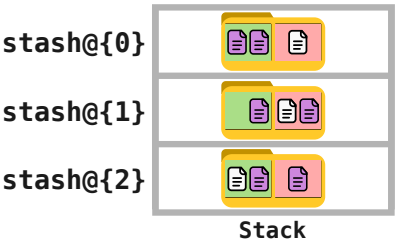
```
git stash -p
```

List all stashes

```
git stash list
```

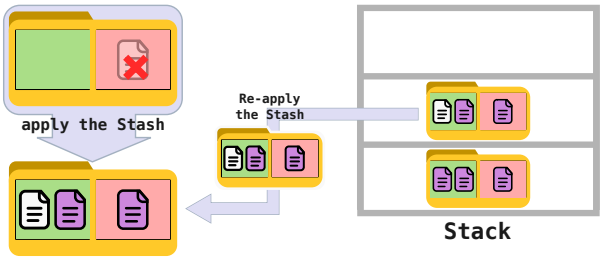
The stashes are named folowing the structure

stash@{#}



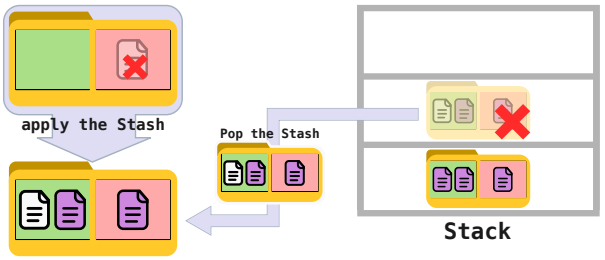
Re-apply the Stash without deleting it

```
git stash apply
```



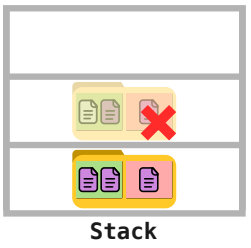
Apply the stash and drop the stash

```
git stash pop
```



Drop a Stash

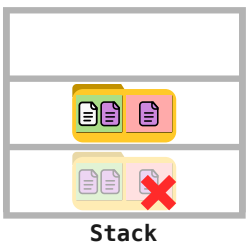
```
git stash drop
```



Drop a specific Stash

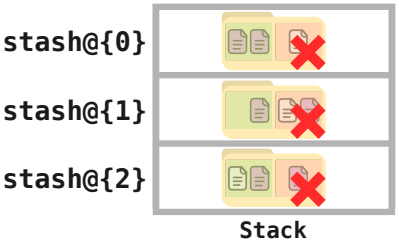
```
git  stash drop stash@{#}
```


```
$ git stash drop stash@{1}
```



Show a stash info
```
git stash show stash@{#}
```

clear all stashes
```
git stash clear
```



Debugging

Blame

The git blame command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code

Show information about changes made in the file <file>
```
git blame <file>
```


```
$ git blame readme.md
```

Show information about changes made in the file <file> before a specified commit
```
git blame <commit-ID> <file>
```


```
$ git blame 9be1074c readme.md
```

Bisect

The bisect tool is a helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search

1. First bisect command must be started,
   ```
   git bisect start
   ```
2. Then you tell the system that the current commit you're on is broken
   ```
   git bisect bad
   ```
3. Then, you must tell bisect when the last known good state was
   ```
   git switch <commit-ID>
   git bisect good
   ```
   Or just
   ```
   git bisect good <commit-ID>
   ```
4. At this point, Git know how many commits are between the commit marked as the last good commit and the current bad version, and start asking you by specific commits to finally find in which commit the bug was introduced.
   ```
   git bisect [good or bad]
   ```
5. Once the bugged commit is individualized you must reset bisect to finish and go back to the normal environment.
   ```
   git bisect reset
   ```

References

- Laster, B. (2016). Professional git. John Wiley & Sons.