

1. Create a repository

A repository is usually used to organize a single project.

Repositories can contain folders and files, images, videos, spreadsheets and data sets.

We recommend including a README, or a file with information about your project.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner

Repository name *

sallychenuni

 /

hello-world

Great repository names are short and memorable. Need inspiration? How about [upgraded-octo-enigma?](#)

Description (optional)

First Repository

☒ Public

Anyone can see this repository. You choose who can commit.

☐ Private

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☒ Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: None

Add a license: None

Create repository

2. Create a Branch

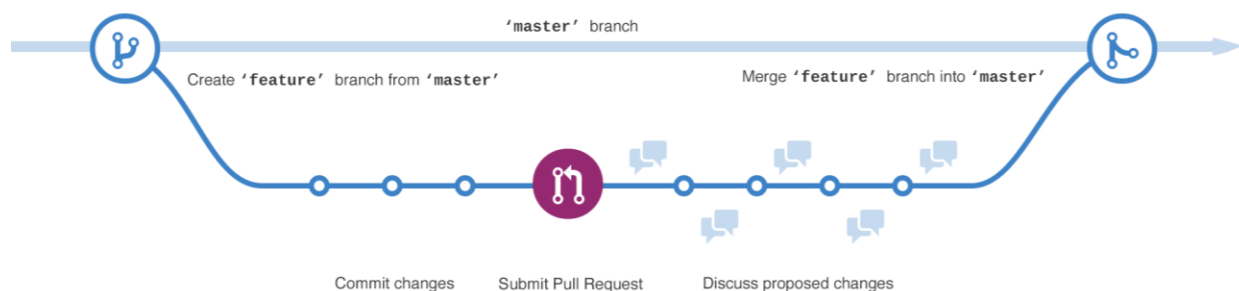
Branching is the way to work on different version of a repository at one time.

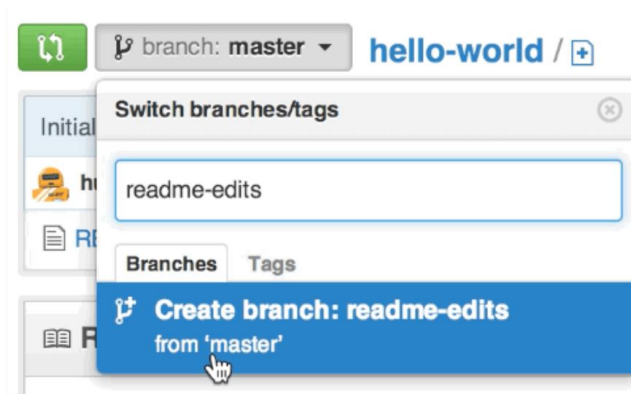
By default, your repository has one branch named `master` which is considered to be the definitive branch. We use branches to experiment and make edits before committing them to master.

When you create a branch off the master branch, you are making a copy, or snapshot of master as it was at that point in time. If someone else made changes to the master branch while you were working on your branch, you could pull in those updates.

A new branch is called `feature`, because we are doing 'feature work' on this branch.

Below is the journey that feature takes before it's merged into master






3. Make and commit changes

You are on the code review for your `readme-edit` branch, which is a copy of master. On GitHub, saved changes are called *commits*. Each commit has an associated *commit message*, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you've done and why.

Make and commit changes

1. Click the `README.md` file.
2. Click the  pencil icon in the upper right corner of the file view to edit.
3. In the editor, write a bit about yourself.
4. Write a commit message that describes your changes.
5. Click **Commit changes** button.

These changes will be made to just the README file on your `readme-edits` branch, so now this branch contains content that is different from master.

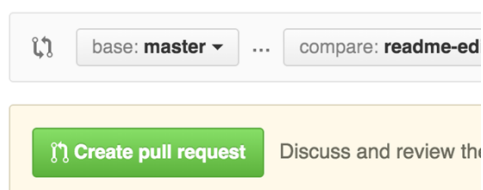
4. Open a Pull Request

Now that you have changes in a branch off of master, you can open a pull request. Pull Requests are the heart of collaboration on GitHub. When you open a *pull* request, you are proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch.

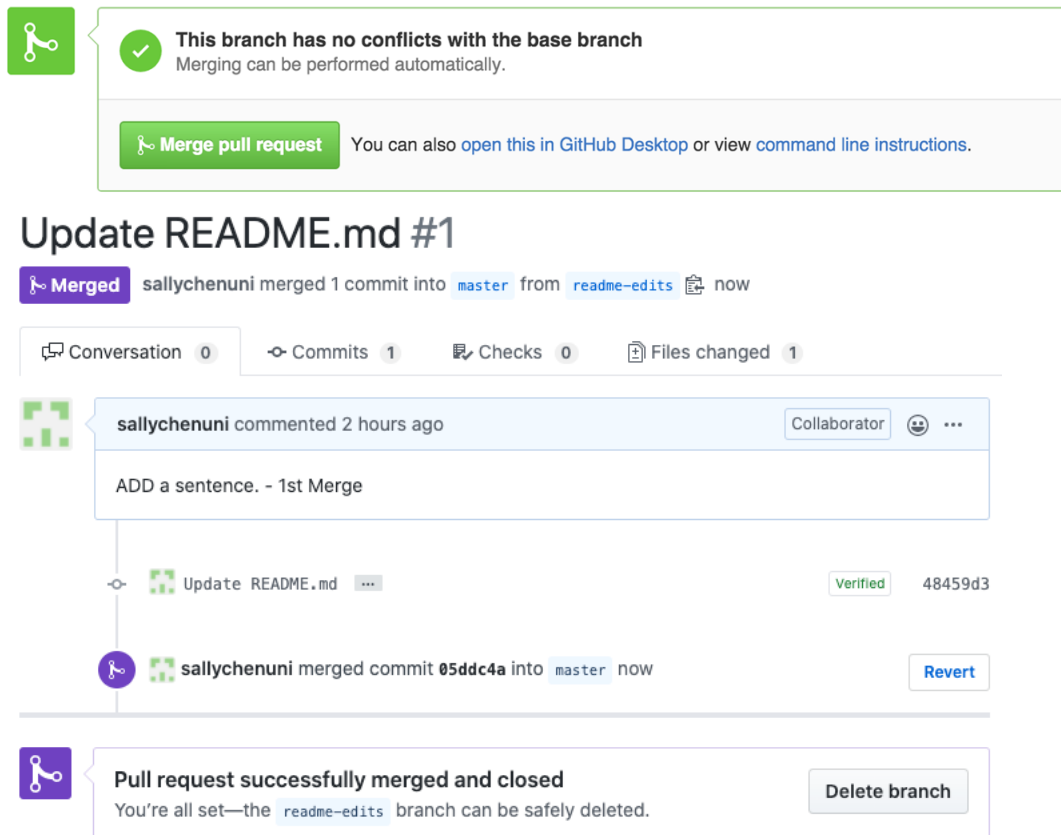
Pull requests show diffs, or differences of the content from both branches. The changes, additions and subtractions are shown in green and red.

As soon as you make a commit, you can open a pull request and start a discussion. By using GitHub's @mention system (You can mention a person or [team](#) on GitHub by typing @ plus their username or team name. This will trigger a notification and bring their attention to the conversation.) People will receive notification.

@github/support What do you think about these updates?



5. Merge Your Pull Request and Delete Branch



The screenshot shows a GitHub pull request interface. At the top, a green box with a checkmark icon contains the text: "This branch has no conflicts with the base branch" and "Merging can be performed automatically." Below this is a green button labeled "Merge pull request" and a link: "You can also open this in GitHub Desktop or view command line instructions." Below the green box is the title "Update README.md #1" and a status bar indicating "Merged" by sallychenuni, merged 1 commit into master from readme-edits now. Below the status bar are tabs for Conversation (0), Commits (1), Checks (0), and Files changed (1). The main content area shows a comment by sallychenuni from 2 hours ago: "ADD a sentence. - 1st Merge". Below the comment is a commit history showing "Update README.md" with a verified commit hash 48459d3. Below the commit history is a message indicating the pull request was successfully merged and closed, with a button to "Delete branch".

1. what is ssh?

With SSH keys, you can connect to GitHub without supplying your username or password at each visit. → Done

Private key, public key

Each SSH key pair includes two keys:

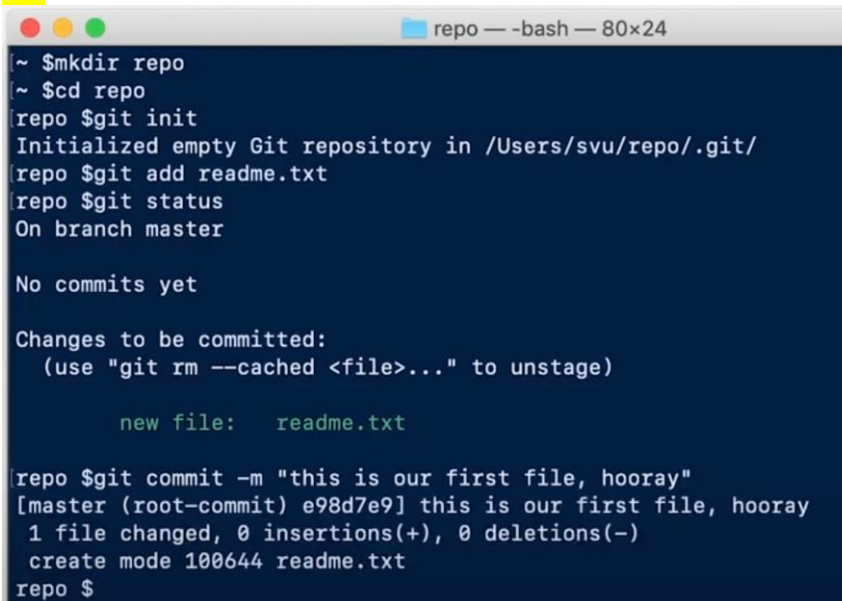
- A public key that is copied to the SSH server(s). Anyone with a copy of the public key can encrypt data which can then only be read by the person who holds the corresponding private key. Once an SSH server receives a public key from a user and considers the key trustworthy, the server marks the key as authorized in its authorized keys file. Such keys are called authorized keys.
- A private key that remains (only) with the user. The possession of this key is proof of the user's identity. Only a user in possession of a private key that corresponds to the public key at the server will be able to authenticate

successfully. The private keys need to be stored and handled carefully, and no copies of the private key should be distributed. The private keys used for user authentication are called identity keys.

Cited: <https://www.ssh.com/ssh/public-key-authentication>

2. Init a repo in github

init

A terminal window titled 'repo — -bash — 80x24' showing the following commands and output:

```
~ $mkdir repo
~ $cd repo
repo $git init
Initialized empty Git repository in /Users/svu/repo/.git/
repo $git add readme.txt
repo $git status
On branch master

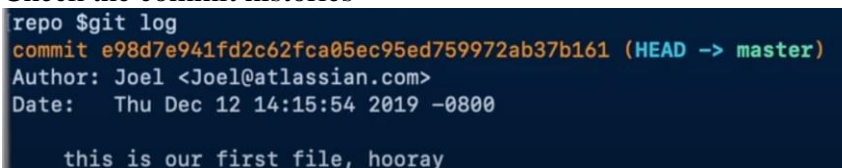
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   readme.txt

repo $git commit -m "this is our first file, hooray"
[master (root-commit) e98d7e9] this is our first file, hooray
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 readme.txt
repo $
```

Check the commit histories

A terminal window showing the output of the 'git log' command:

```
repo $git log
commit e98d7e941fd2c62fca05ec95ed759972ab37b161 (HEAD -> master)
Author: Joel <Joel@atlassian.com>
Date:   Thu Dec 12 14:15:54 2019 -0800

    this is our first file, hooray
```

3. Clone a repo

git clone <url>

4. add a readme.md edit the file

git add readme.md

git commit -m 'add readme.md file'

5. Submit the file

Comment standard

Explanatory text explaining what has been changed and why the change was necessary. Write in the imperative mood (present tense)

6. Create a branch called dev

7. Checkout the branch

checkout

8. Make changes

9. Submit changes

Exercise - 12 May 2020

10. Readme.md template

README file is normally the first entry point to your code. It should tell people why they should use your module, how they can install it and how they can use it. People can use your module without ever having to look at its code.

Markdown format:

<https://help.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax>

11. Gitflow

Gitflow is an abstract idea of a Git workflow.

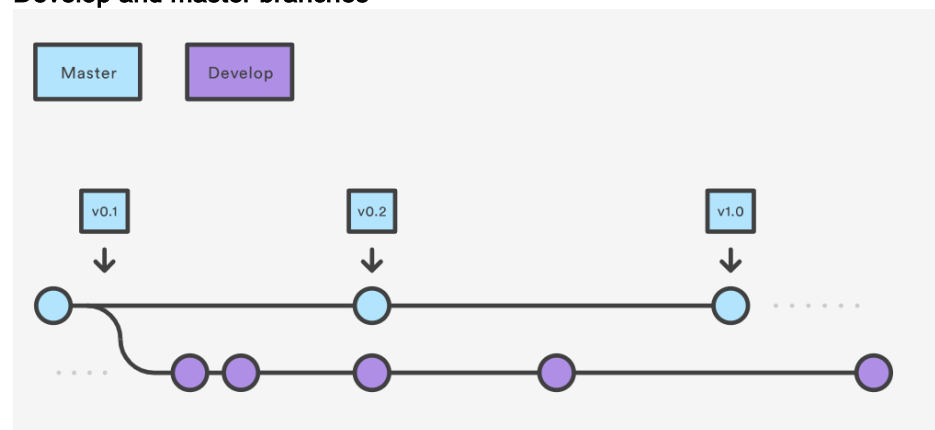
Gitflow workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.

Gitflow is ideally suited for projects that have a scheduled release cycle. It assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it used individual branches for preparing, maintaining and recording releases.

Gitflow dictates what kind of branches to set up and how to merge them together. The git-flow toolset is an actual command line tool that has an installation process. On OSX systems, you can execute `brew install git-flow`.

After installing git-flow you can use it in your project by executing `git flow init`. Git-flow is a wrapper around Git. The `git flow init` command is an extension of the default `git init` command and doesn't change anything in your repository other than creating branches for you.

Develop and master branches



Instead of a single master branch, this workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features.

The first step is to complement the default master with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

```
git branch develop
git push -u origin develop
```

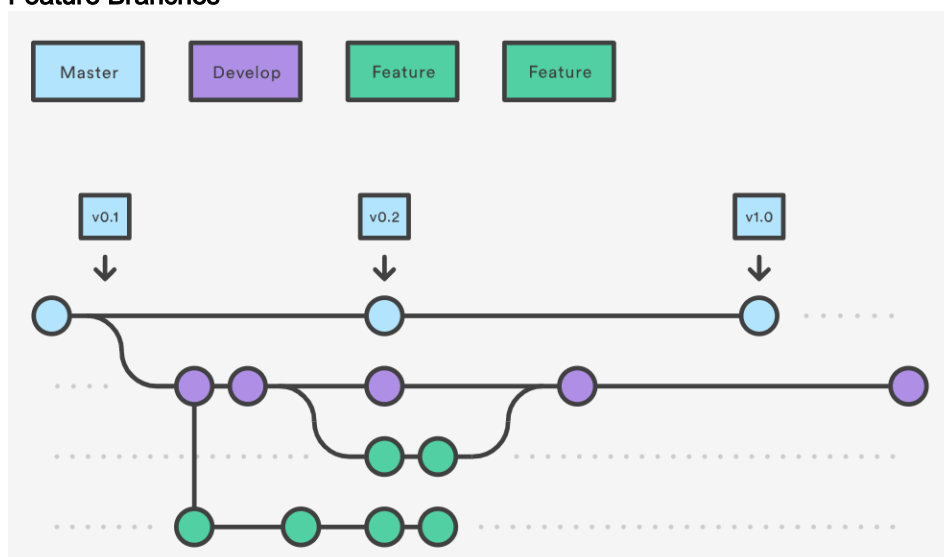
When using the git-flow extension library, executing `git flow init` on an existing repo will create the develop branch:

```
$ git flow init
Initialized empty Git repository in ~/project/.
No branches exist yet. Base branches must be created.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

$ git branch
* develop
master
```

Feature Branches



Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. Feature branches use develop as their parent branch. When a feature is complete, it gets merged back into develop. Features should never interact directly with master.

Feature branches combined with the develop branch is the Feature Branch Workflow. Feature branches are generally created off the latest develop branch.

Creating a feature branch

Without the git-flow extensions:

```
git checkout develop
git checkout -b feature_branch
```

When using the git-flow extension:

```
git flow feature start feature_branch
```

Continue your work and use Git like you normally would.

Finishing a feature branch

When you're done with the development work on the feature, the next step is to merge the `feature_branch` into `develop`.

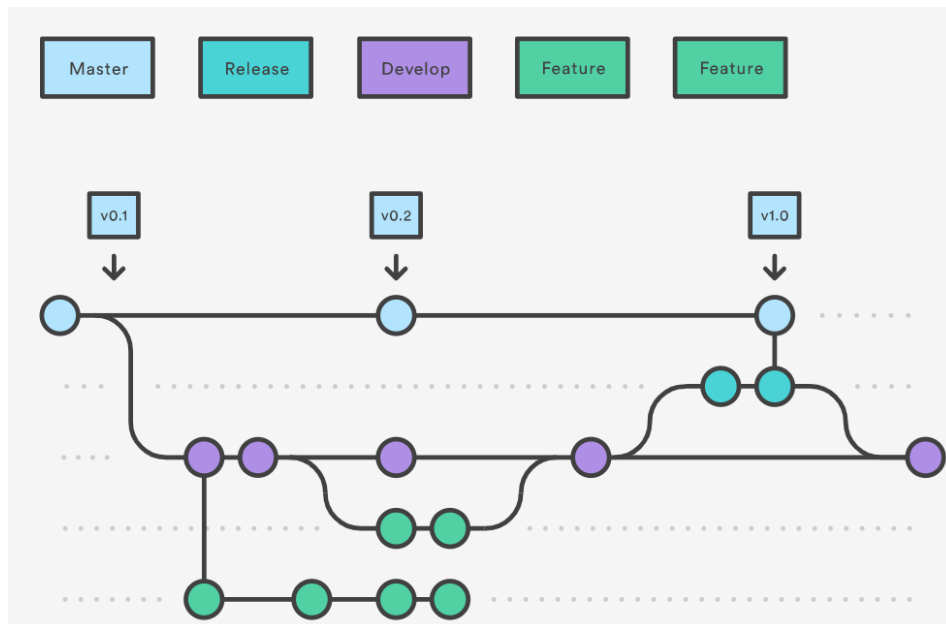
Without the git-flow extensions:

```
git checkout develop
git merge feature_branch
```

Using the git-flow extensions:

```
git flow feature finish feature_branch
```

Release Branches



Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop.

Creating this branch starts the next release cycle, so no new features can be added after this point (only bug fixes, documentation generation and other release-oriented tasks should go in this branch.)

Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare release makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g. it's easy to say, "This week we're preparing for version 4.0," and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation. Like feature branches, release branches are based on the develop branch.

A new release branches can be created using the following methods:

Without the git-flow extensions:

```
git checkout develop
git checkout -b release/0.1.0
```

When using the git-flow extensions:

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```


Once the release is ready to ship, it will get merged into master and develop, then the release branch will be deleted. It is important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organisation stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

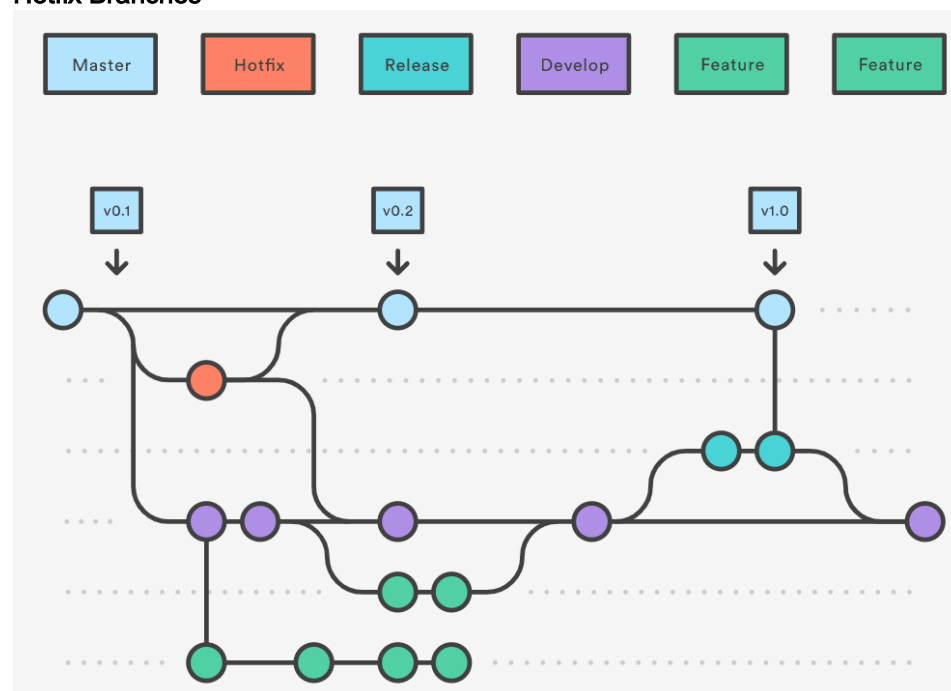
Without the git-flow extensions:

```
git checkout master
git merge release/0.1.0
```

Or with the git-flow extension:

```
git flow release finish '0.1.0'
```

Hotfix Branches



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

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle.

Think maintenance branches as ad hoc release branches that work directly with master.

A hotfix branch can be created using the following methods:

Without the git-flow extensions:

```
git checkout master  
git checkout -b hotfix_branch
```

When using the git-flow extensions:

```
$ git flow hotfix start hotfix_branch
```

Similar to finishing a release branch, a hotfix branch gets merged into both master and develop.

```
git checkout master  
git merge hotfix_branch  
git checkout develop  
git merge hotfix_branch  
git branch -D hotfix_branch
```

```
$ git flow hotfix finish hotfix_branch
```

Examples

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a master branch.

```
git checkout master
git checkout -b develop
git checkout -b feature_branch
# work happens on feature branch
git checkout develop
git merge feature_branch
git checkout master
git merge develop
git branch -d feature_branch
```

In addition to the feature and release flow, a hotfix example is as follows:

```
git checkout master
git checkout -b hotfix_branch
# work is done commits are added to the hotfix_branch
git checkout develop
git merge hotfix_branch
git checkout master
git merge hotfix_branch
```

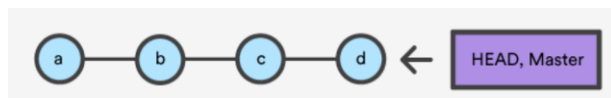
The overall flow of Gitflow is:

- A develop branch is created from master
- A release branch is created from develop
- Feature branches are created from develop
- When a feature is complete it is merged into the develop branch
- When the release branch is done it is merged into develop and master
- If an issue in master is detected a hotfix branch is created from master
- Once the hotfix is complete it is merged to both develop and master

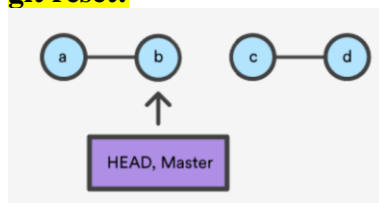
Cited: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

12. Git reset head vs. git checkout

Command	Scope	Common use cases
git reset	Commit-level	Discard commits in a private branch or throw away uncommitted changes
git reset	File-level	Unstage a file
git checkout	Commit-level	Switch between branches or inspect old snapshots
git checkout	File-level	Discard changes in the working directory
git revert	Commit-level	Undo commits in a public branch
git revert	File-level	(N/A)

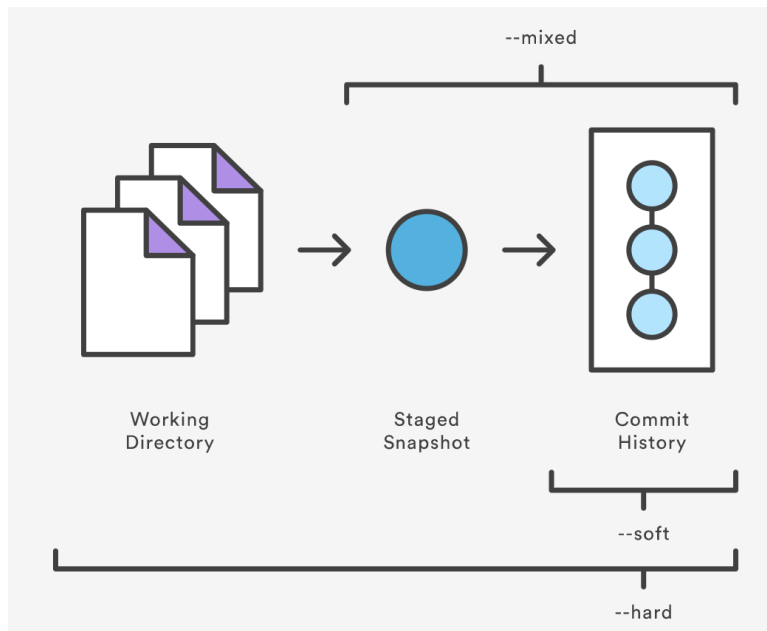


git reset:



Three trees of Git (three modes):

The Commit Tree (HEAD, done git commit), The Staging Index (done git add, staged), Working Directory (not staged)



git reset --hard

This is the most direct, DANGEROUS, and frequently used option.

Any pending work that was hanging out in the **staging index and working directory** will be lost. This data loss cannot be undone.

git reset --mixed / git reset

This is the default operating mode.

The **Staging Index** has been reset and the pending changes have been moved into the Working Directory.

git reset --soft

The Staging Index and the Working Directory are left untouched. A soft reset will only reset the **Commit History**. **?Only today's**

Never use `git reset <commit>` after `<commit>` have been pushed to a public repository, use `git revert` instead. A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit.

git reset <mode> <file>

Unstage specific file to working directory and leave the working directory unchanged.

git reset HEAD~2 foo.py

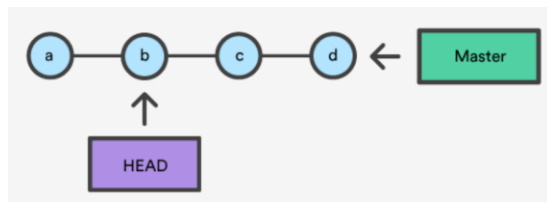
git reset <mode> <commit>

Move the current branch tip backward to `<commit>`, reset the staging area to match, but leave the working directory alone. All changes made since `<commit>` will reside in the working directory.

git reset HEAD~2

move branch backwards by two commits.

git checkout:

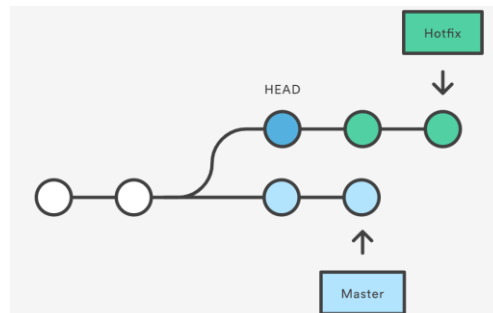
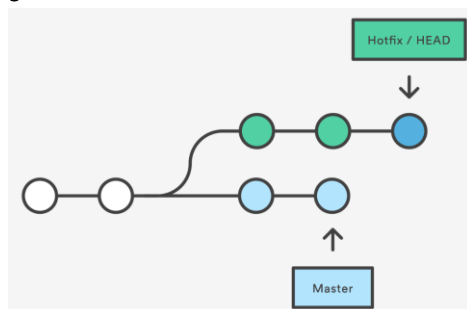


git checkout <branch>

Move HEAD to a different branch and update the working directory (local folder) to match. Since this has the potential to overwrite local changes, Git forces you to commit or stash any changes in the working directory that will be lost during the checkout operation

git checkout doesn't move any branches around.

git checkout HEAD~2



Useful for quickly inspecting an old version of your project.

However, since there is no branch reference to the current HEAD, this puts you in a detached HEAD state. This can be dangerous if you start new commits because there will be no way to get back to them after you switch to another branch.

For this reason, you should always create a new branch before adding commits to a detached HEAD.

git checkout <file>

Checking out a file is similar to using git reset with a file path, except it updates the *working directory* instead of the stage.

Unlike the commit-level version of this command, this does not move the HEAD reference, which means that you won't switch branches.

git checkout HEAD~2 foo.py

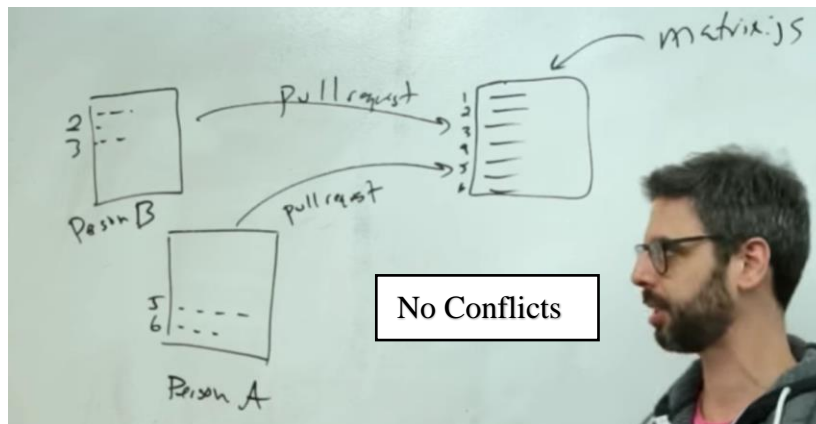
Used to inspect old version of a project, but the scope is limited to the specified file. Reverting to the old version of that file and removing all of the subsequent changes to the file.

git checkout HEAD foo.py has the effect of discarding unstaged changes to foo.py. This is similar behavior to git reset HEAD --hard, but it operates only on the specified file.

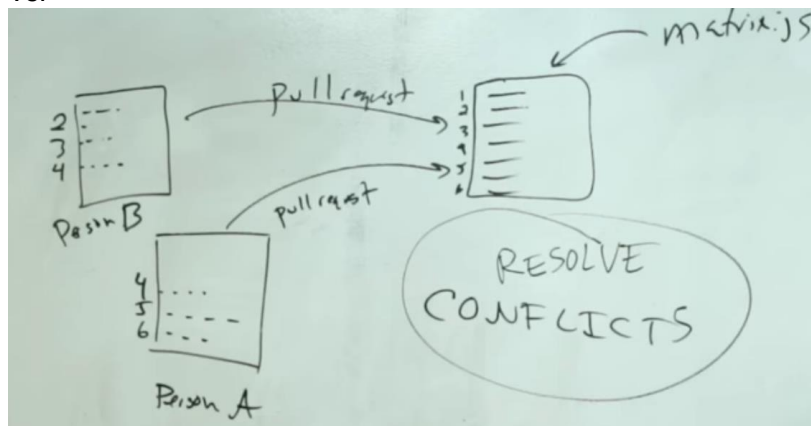
Cited: <https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting>

13. How to deal with conflict

Examples:



Vs.



Type1: Git fails to start the merge

This occurs because git knows there are changes in either your working directory or staging area that could be written over by the files that you are merging in. If this happens, there are no merge conflicts in individual files. You need to modify or stash the files it lists and then try to do a `git pull` again.

The error messages are as follows:

```
error: Entry '<fileName>' not up to date. Cannot merge. (Changes in working directory)
```

or

```
error: Entry '<fileName>' would be overwritten by merge. Cannot merge. (Changes in staging area)
```

Type2: Git fails during the merge

A failure during a merge indicates a conflict between the current local branch and the branch being merged. This occurs because you have committed changes that are in conflict with someone else's committed changes.

The error messages are as follows:

```
CONFLICT (content): Merge conflict in <fileName>
Automatic merge failed; fix conflicts and then commit the result.
```

How to identify merge conflicts

If your merge failed to even start, there will be no conflicts in files.

If Git finds conflicts during the merge, it will list all files that have conflicts after the error message.

You can also check on which files have merge conflicts by doing a `'git status'`.

```

$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt

```

The output from git status indicates that there are unmerged paths due to a conflict. We can use cat command to put out the contents of the merge.txt file.

```

$ cat merge.txt
<<<<<< HEAD
this is some content to mess with
content to append
=====
totally different content to merge later
>>>>>> new_branch_to_merge_later

```

The ===== line is the "centre" of the conflict. All the content between the centre and the <<<<<< HEAD is content that exists in the current branch master which you are trying to merge into.

All content between the center and >>>>>> new_branch_to_merge_later is content that is present in our merging branch.

How to resolve merge conflicts using the command line

The most direct way to resolve a merge conflict is to edit the conflicted file. Open the merge.txt file in your favourite editor.

For example, lets simply remove all the conflict dividers.

The modified merge.txt content should then look like:

```

this is some content to mess with
content to append
totally different content to merge later

```

Once the file has been edited, use git add merge.txt to stage the new merged content and git commit to finalize the merge. Git will see that the conflict has been resolved.

