
epam
Version Control Systems
Git + Git Worklows
git
Author: Aleksandr Mazurenko
2019

Agenda

- Version Control Systems
  - Introduction to Version Control Systems
  - Benefits of Version Control Systems
  - Common terms
  - VCS classification

- GIT Basic
- GIT Workflows
- GitHub - Overview

Version Control Systems

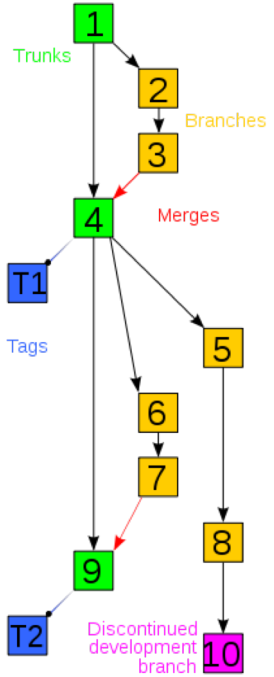
Sonatype Nexus

What is Version control System?

Version control system is a system that records changes to a file or set of files over time so that you can recall specific versions later.

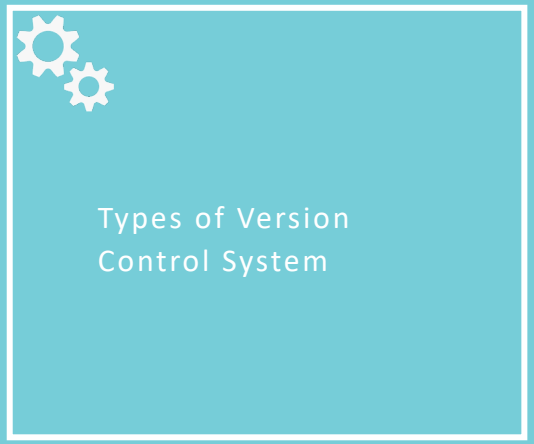
Widely used in any sphere where keeping every version of an document or layout is required.

It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.



Version Control Systems Types:

- Local Version Control Systems
- Centralized Version Control Systems
- Distributed Version Control Systems



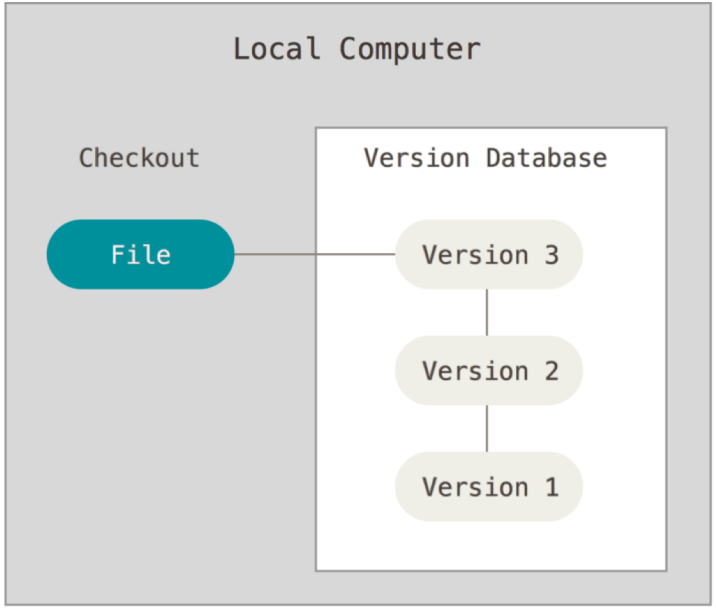
Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever).

This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

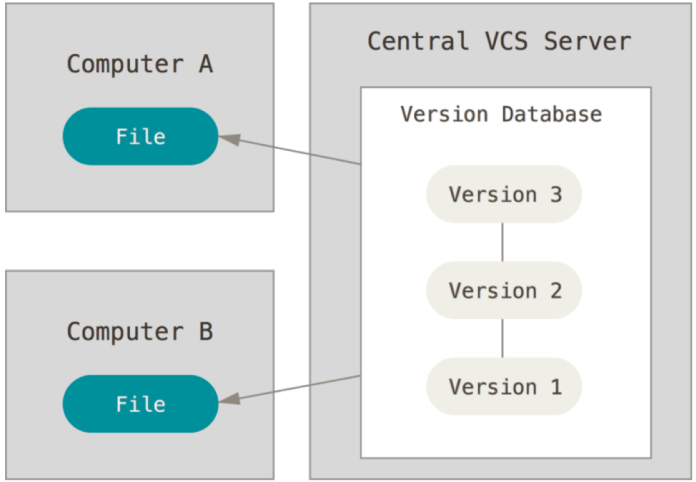
To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

One of the most popular VCS tools was a system called RCS(Revision Control System), which is still distributed with many computers even today.



Centralized Version Control Systems

These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

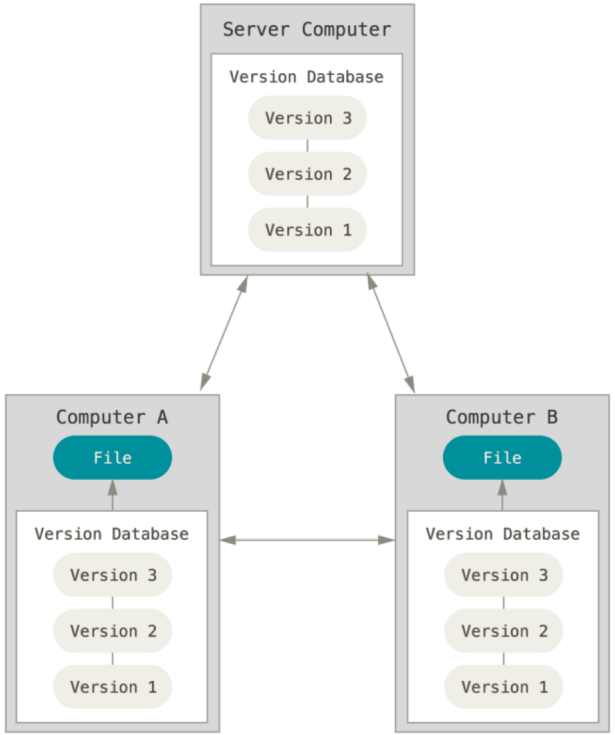
However, this setup also has some serious downsides - single point of failure:

- If server goes down nobody can collaborate at all or save versioned changes to anything they're working on
- If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.



Benefits

- Collaboration
- Change management
- Tracking ownership
- Tracking evolution
- Branching
- Continuous integration.

Git

- http://git-scm.com/downloads

Check the version using the following command to verify that Git is installed and successfully operational:

git --version

- **Initial Configuration - Establishing User Identity**

Once you have a distribution of Git installed for your platform, identify yourself with a username and email address. This is strictly to credit your efforts on commits and should be done for each machine on which you'll be using Git.

```
# Mandatory
git config --global user.name "Jordan McCullough"
git config --global user.email jmccullough@github.com

# Optional
git config --global color.ui "auto"
```

Getting a Git Repository

You typically obtain a Git repository in one of two ways:

- You can take a local directory that is currently not under version control, and turn it into a Git repository, or
- You can *clone* an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

Initializing a Repository in an Existing Directory

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory. If you've never done this, it looks a little different depending on which system you're running:

```
## for Linux:
      $ cd /home/user/my_project

## for macOS:
      $ cd /Users/user/my_project

## for Windows:
      $ cd /c/user/my_project
```

```
## and type:
      $ git init
```

This creates a new subdirectory named ***.git*** that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. (See Git Internals for more information about exactly what files are contained in the ***.git*** directory you just created.)

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few *git add* commands that specify the files you want to track, followed by a git commit:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

At this point, you have a Git repository with tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is *git clone <url>*

```
## For example
$ git clone https://github.com/libgit2/libgit2
```

That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2directory that was just created, you'll see the project files in there, ready to be worked on or used.

Git has a number of different transfer protocols you can use. The previous example uses the https://protocol, but you may also see **git://** or **user@server:path/to/repo.git**, which uses the SSH transfer protocol. Getting Git on a Server will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

There are the following main states that your files can reside in:

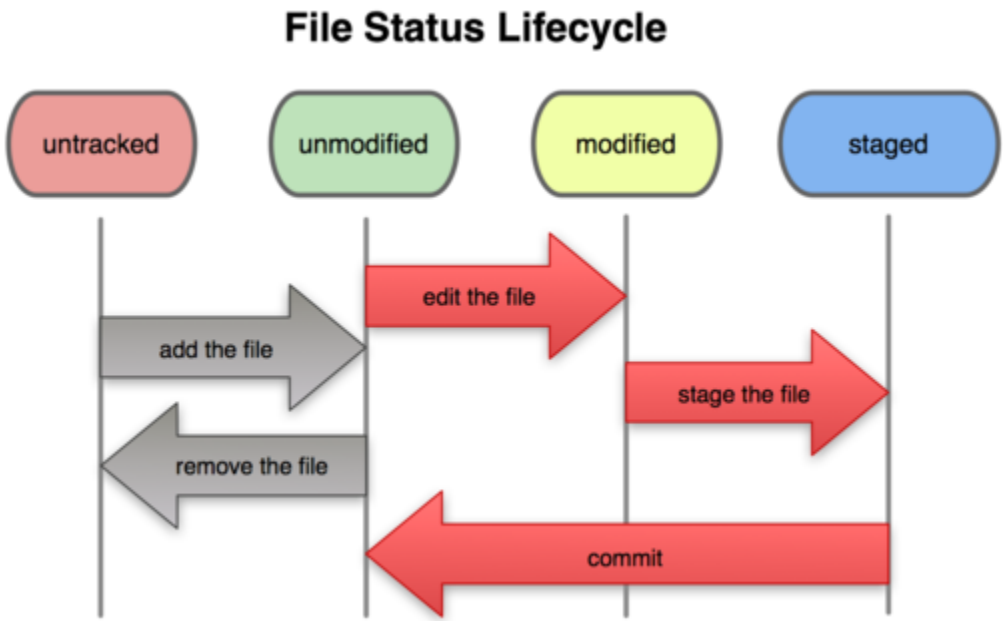
- *untracked* – the file just created or removed

==========

- *unmodified (committed)* - means that the data is safely stored in your local database.
- *modified* - means that you have changed the file but have not committed it to your database yet.
- *staged* - means that you have marked a modified file in its current version to go into your next commit snapshot.

==========

- *ignored* - files which are listed in *.gitignore.* Git doesn't track that files at all.



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
On branch master Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here.

Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server.

https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

```
$ cat .gitignore
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir /TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the ***git log*** command.

These examples use a very simple project called "simplegit". To get the project, run

```
$ git clone https://github.com/schacon/simplegit-progit
```

When you run git log in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

     changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700

     removed unnecessary test
```

https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Undoing Things

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 -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Unstaging a Staged File

```
$ git add *
$ git status
On branch master
Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

    renamed: README.md -> README
    modified: CONTRIBUTING.md
```

https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

Unstaging a Staged File

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M       CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

    renamed: README.md -> README

Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

    modified: CONTRIBUTING.md
```

https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

Working with remotes

Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more

Remote repositories can be on your local machine.

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity...
done.
$ cd ticgit

$ git remote
origin

$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Adding new remotes.

```
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run *git fetch <remote>*:

```
$ git fetch origin
$ git fetch pb
```

You can use the ***git pull*** command to automatically ***fetch*** and then ***merge*** that remote branch into your current branch

https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

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>.***

```
$ git push origin master
```

Inspecting, renaming, removing remotes

To inspect remote you should run: ***git remote show <remote>***

For renaming : ***$ git remote rename <remote> <newRemoteName>***

To remove remote completely: ***$ git remote remove <remote>***

https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Working with Branches

```
## Creating new branch (from default)
  $ git checkout -b <branchName>

## Creating new branch from specific branch ('develop' in example)
  $ git checkout -b <branchName> develop

## Deleting branch
  $ git branch -d <branchNameToDelete>

## Switching to branch
  $ git checkout <branchName>

## Pushing branch to remote
  $ git push origin <branchName>
```

https://git-scm.com/book/en/v2/Git-Basics-Tagging

Working with Tags

```
  $ git tag
  v1.0
  v2.0

## Search
  $ git tag -l "v1.8.5*"
  v1.8.5
  v1.8.5-rc0

## Annotated tag
  $ git tag -a v1.4 -m "my version 1.4”

 ## Tagging later (specific commit)
  $git tag -a v1.2 9fceb02

## Sharing tags
  $ git push origin v1.5
  $ git push origin –tags

## Checking out tag
  $ git checkout v2.0.0
  $ git checkout -b version2 v2.0.0
  Switched to a new branch 'version2'
```

https://git-scm.com/book/en/v2/Git-Basics-Tagging

Aliases

```
## Examples
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git br
```

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own unstage alias to Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

This executes external command (pay attention please , you should use '!'):

```
$ git config --global alias.visual '!gitk'
```

https://git-scm.com/book/en/v2/Git-Basics-Git-Aliases

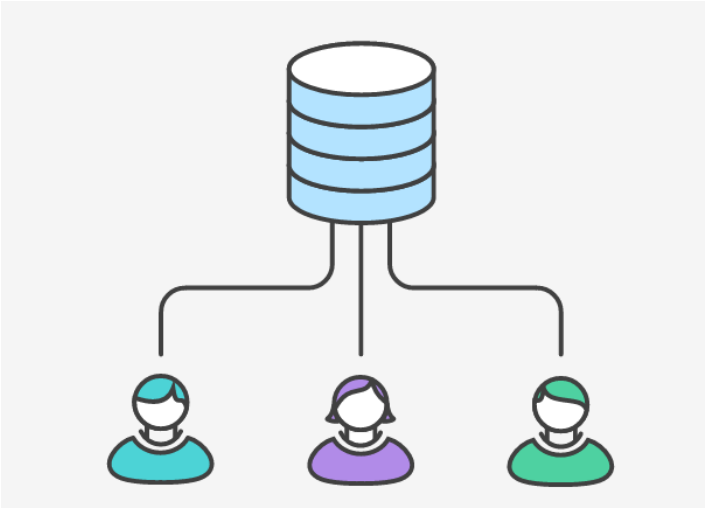
Git WorkFlows

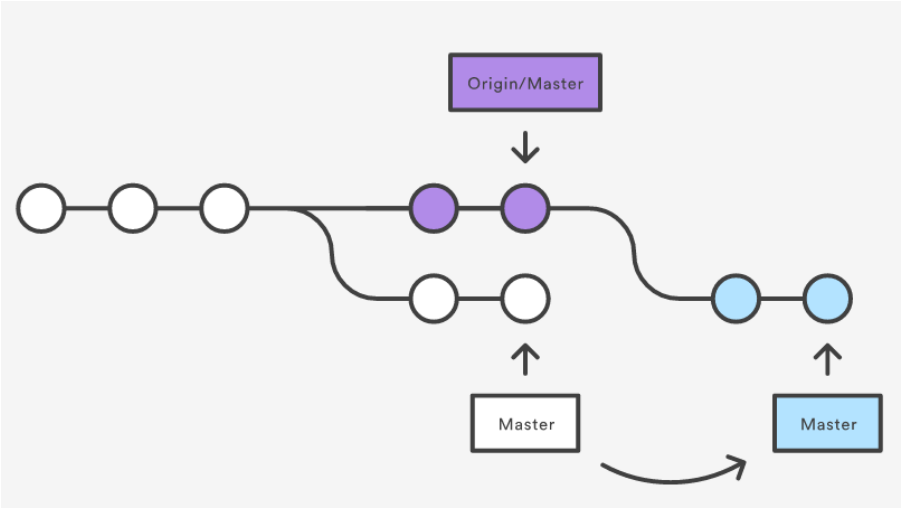
List of Git Workflows

- Centralized Workflow
- Feature Branch Workflow
- Gitflow Workflow
- Forking Workflow

https://www.atlassian.com/git/tutorials/comparing-workflows

GIT WORKFLOW – CENTRALIZED WORKFLOW

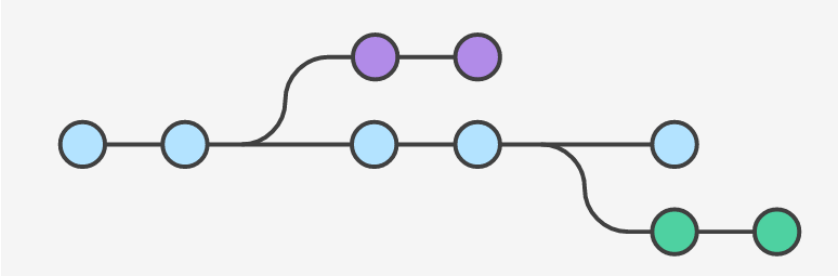




The **Centralized Workflow** is a great Git workflow for teams transitioning from SVN. Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of ***trunk***, the default development branch is called ***master*** and all changes are committed into this branch. This workflow doesn't require any other branches besides ***master***.

https://www.atlassian.com/git/tutorials/comparing-workflows

GIT WORKFLOW – FEATURE BRANCH WORKFLOW



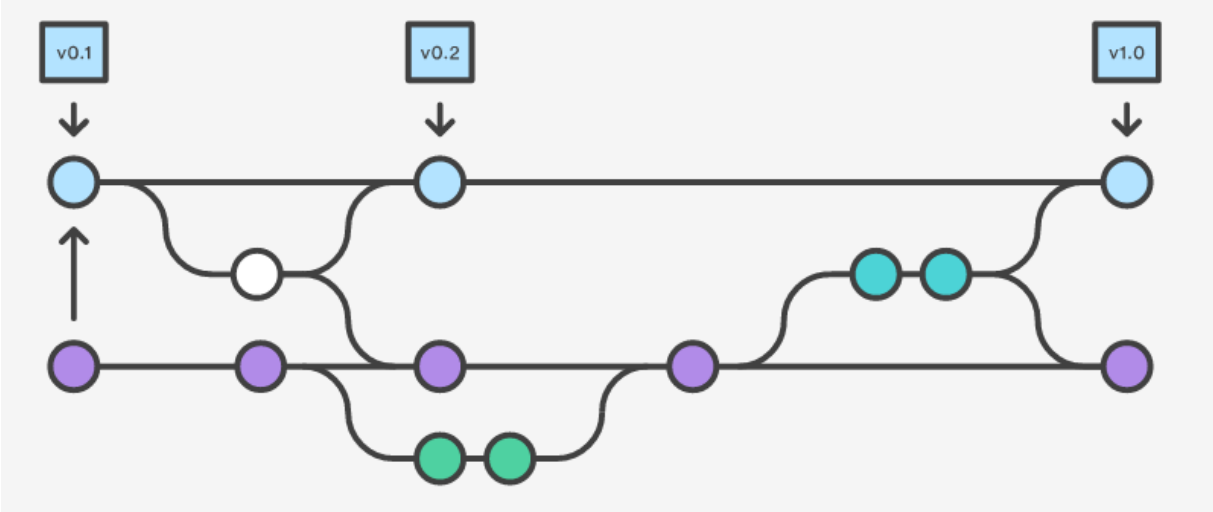
If your team is comfortable with the Centralized Workflow but wants to streamline its collaboration efforts, it's definitely worth exploring the benefits of the **Feature Branch Workflow**. By dedicating an isolated branch to each feature, it’s possible to initiate in-depth discussions around new additions before integrating them into the official project.

Also it allows you to use pull requests

The Feature Branch Workflow is an incredibly flexible way to develop a project. The problem is, sometimes it’s too flexible. For larger teams, it’s often beneficial to assign more specific roles to different branches.

https://www.atlassian.com/git/tutorials/comparing-workflows
https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

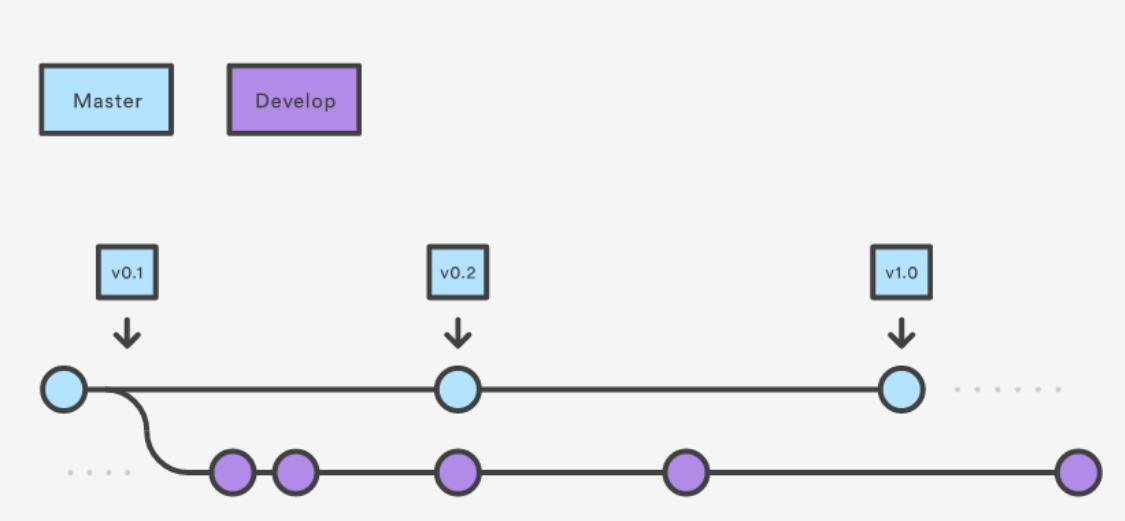
GIT WORKFLOW – GITFLOW WORKFLOW



This workflow doesn't add any new concepts or commands beyond what's required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

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

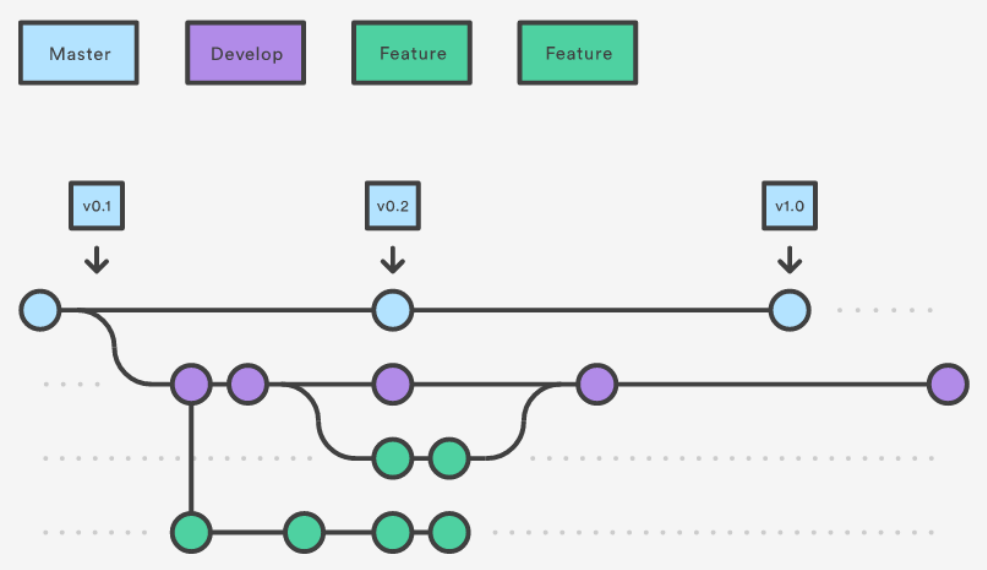
GIT WORKFLOW – GITFLOW WORKFLOW - HISTORY BRANCHES



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. It's also convenient to tag all commits in the master branch with a version number.

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

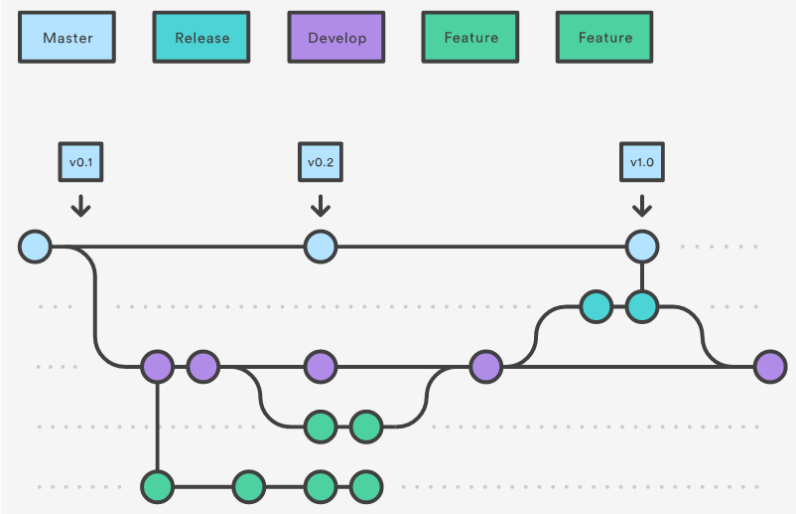
GIT WORKFLOW – GITFLOW WORKFLOW - FEATURE BRANCHES



Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of master, 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.

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

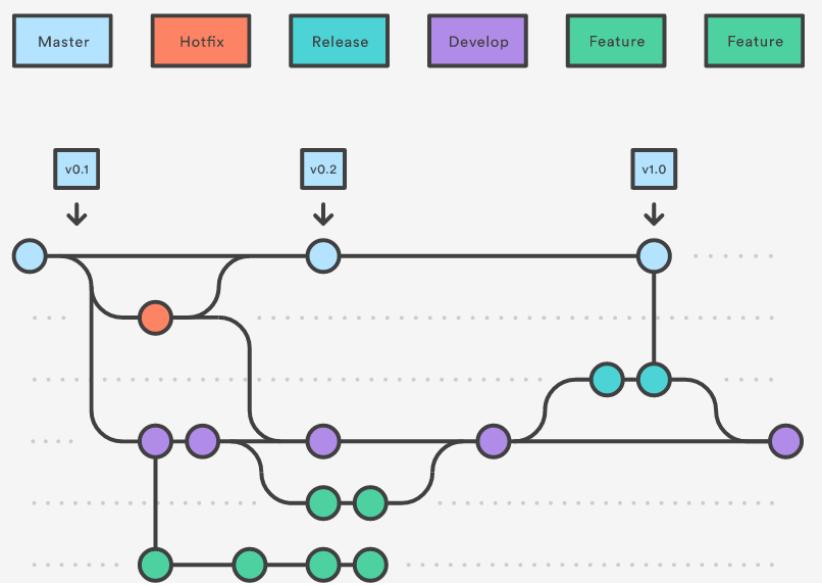
GIT WORKFLOW – GITFLOW WORKFLOW - RELEASE BRANCHES



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 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.

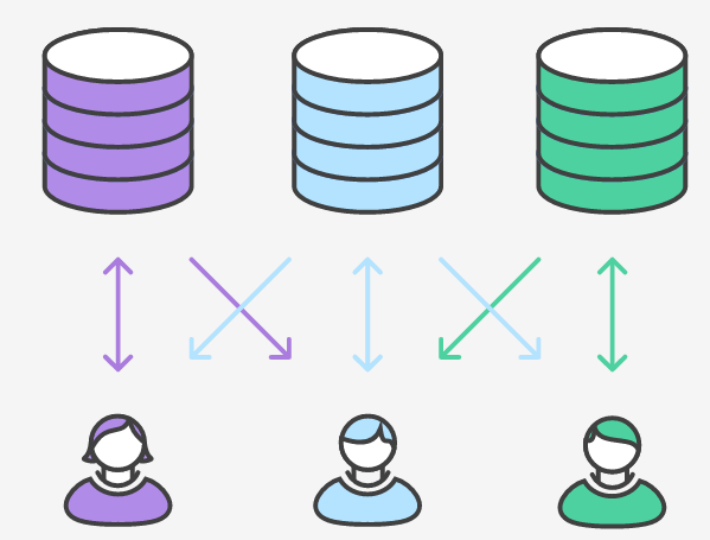
https://www.atlassian.com/git/tutorials/comparing-workflows
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

GIT WORKFLOW – GITFLOW WORKFLOW - MAINTENACE BRANCHES



Maintenance or “hotfix” branches are used to quickly patch production releases. 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 the current release branch), and master should be tagged with an updated version number.

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



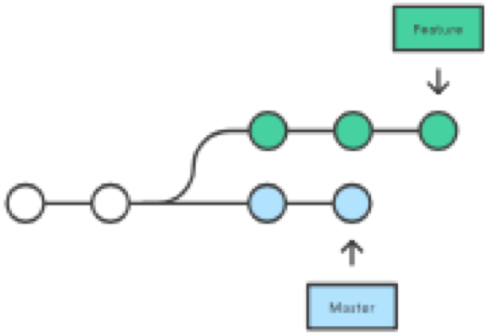
The main advantage of the **Forking Workflow** is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

https://www.atlassian.com/git/tutorials/comparing-workflows
https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

The first thing to understand about ***git rebase*** is that it solves the same problem as ***git merge***. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

A forked commit history



Now, let's say that the new commits in master are relevant to the feature that you're working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

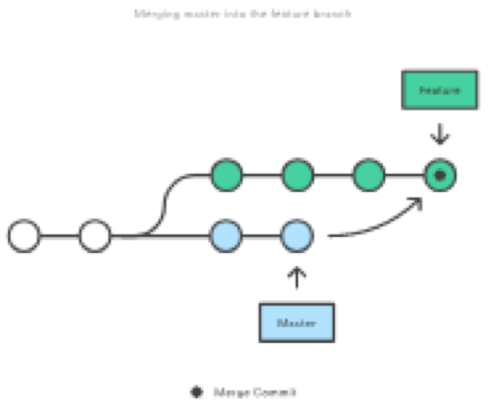
GIT WORKFLOW – MERGE OPTION

The easiest option is to merge the master branch into the feature branch using something like the following:

```
git checkout feature
git merge master
```

Or, you can condense this to a one-liner:

```
git merge master feature
```



This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:

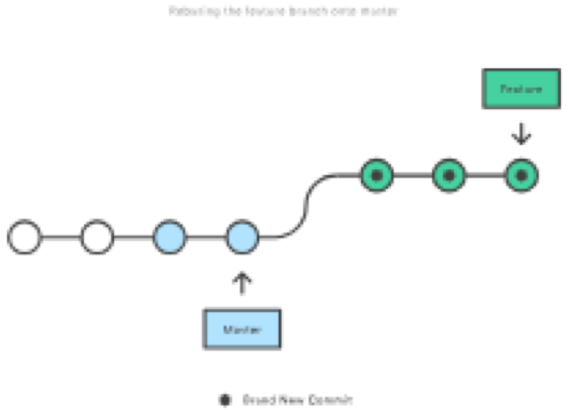
Merging is nice because it’s a *non-destructive* operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).
On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

GIT WORKFLOW – REBASE OPTION

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

```
git checkout feature
git rebase master
```

This moves the entire ***feature*** branch to begin on the tip of the master branch, effectively incorporating all of the new commits in ***master***. But, instead of using a merge commit, rebasing ***re-writes*** the project history by creating brand new commits for each commit in the original branch.



The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by ***git merge***. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like *git log, git bisect*, and *gitk.*

But, there are two trade-offs for this pristine commit history: safety and traceability. If you don't follow the Golden Rule of Rebasing, re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit—you can't see when upstream changes were incorporated into the feature.

GIT WORKFLOW – INTERACTIVE REBASING

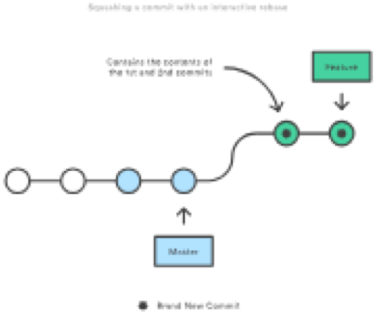
Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch. This is even more powerful than an automated rebase, since it offers complete control over the branch's commit history. Typically, this is used to clean up a messy history before merging a feature branch into master.

To begin an interactive rebasing session, pass the i option to the git rebase command:

```
git checkout feature
git rebase -i master
```

This will open a text editor listing all of the commits that are about to be moved:

```
pick 33d5b7a Message for commit #1
pick 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```



This listing defines exactly what the branch will look like after the rebase is performed. By changing the pick command and/or re-ordering the entries, you can make the branch's history look like whatever you want. For example, if the 2nd commit fixes a small problem in the 1st commit, you can condense them into a single commit with the fixup command:

```
pick 33d5b7a Message for commit #1
fixup 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

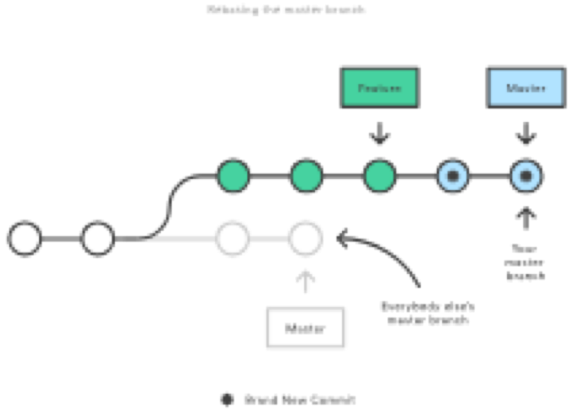
When you save and close the file, Git will perform the rebase according to your instructions, resulting in project history that looks like the following (see picture).

Eliminating insignificant commits like this makes your feature's history much easier to understand. This is something that git merge simply cannot do.

GIT WORKFLOW – GOLD RULE OF REBASING

Once you understand what rebasing is, the most important thing to learn is when *not* to do it. The golden rule of git rebase is to never use it on *public* branches.

For example, think about what would happen if you rebased master onto your feature branch:



The rebase moves all of the commits in master onto the tip of feature. The problem is that this only happened in *your* repository. All of the other developers are still working with the original master. Since rebasing results in brand new commits, Git will think that your master branch's history has diverged from everybody else's.

The only way to synchronize the two master branches is to merge them back together, resulting in an extra merge commit *and* two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch). Needless to say, this is a very confusing situation.

So, before you run git rebase, always ask yourself, "Is anyone else looking at this branch?" If the answer is yes, take your hands off the keyboard and start thinking about a non-destructive way to make your changes (e.g., the ***git revert*** command). Otherwise, you're safe to re-write history as much as you like.

If you try to push the rebased master branch back to a remote repository, Git will prevent you from doing so because it conflicts with the remote master branch. But, you can force the push to go through by passing the --force flag, like so:

```
# Be very careful with this command!
git push --force
```

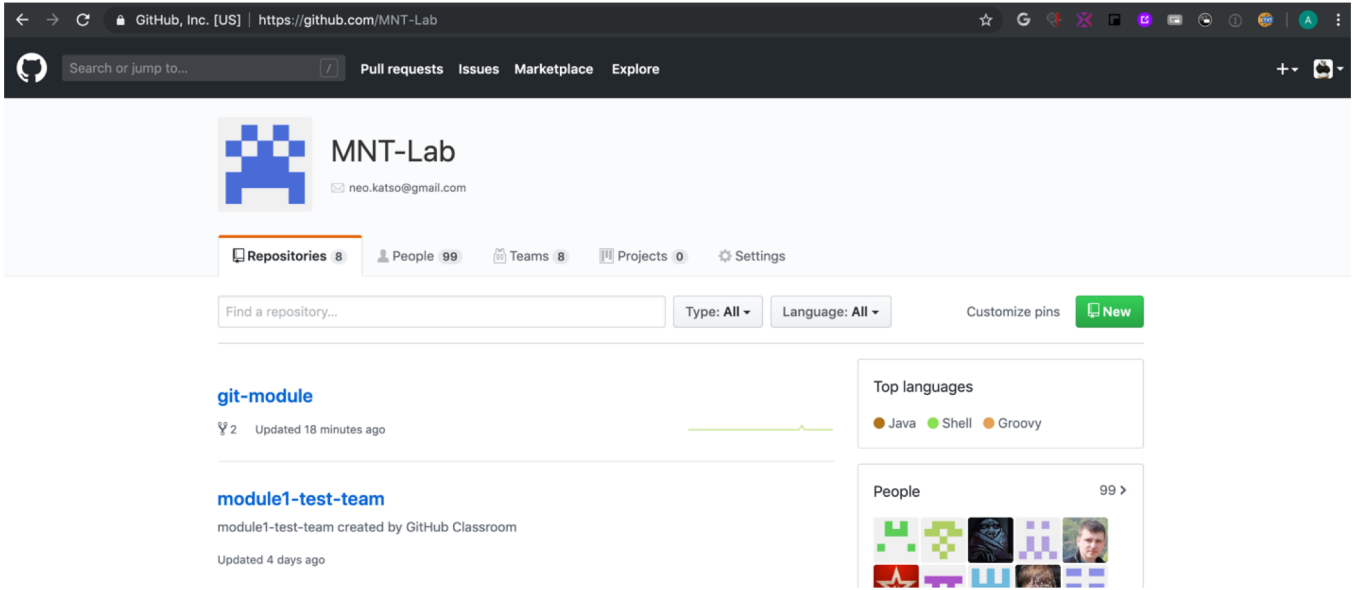
This overwrites the remote master branch to match the rebased one from your repository and makes things very confusing for the rest of your team. So, be very careful to use this command only when you know exactly what you're doing.

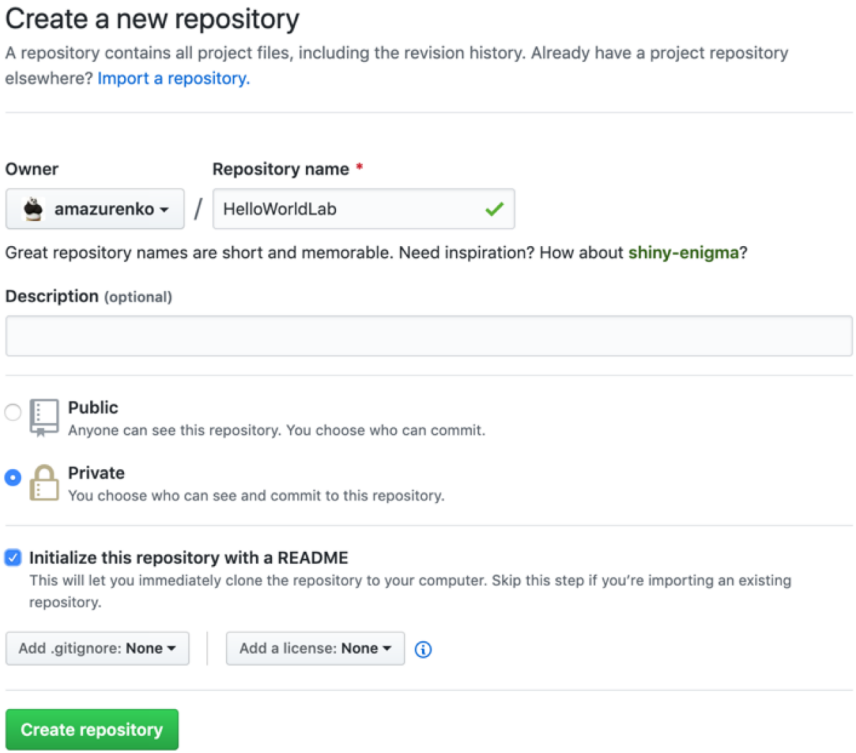
One of the only times you should be force-pushing is when you've performed a local cleanup *after* you've pushed a private feature branch to a remote repository (e.g., for backup purposes). This is like saying, "Oops, I didn't really want to push that original version of the feature branch. Take the current one instead." Again, it's important that nobody is working off of the commits from the original version of the feature branch.

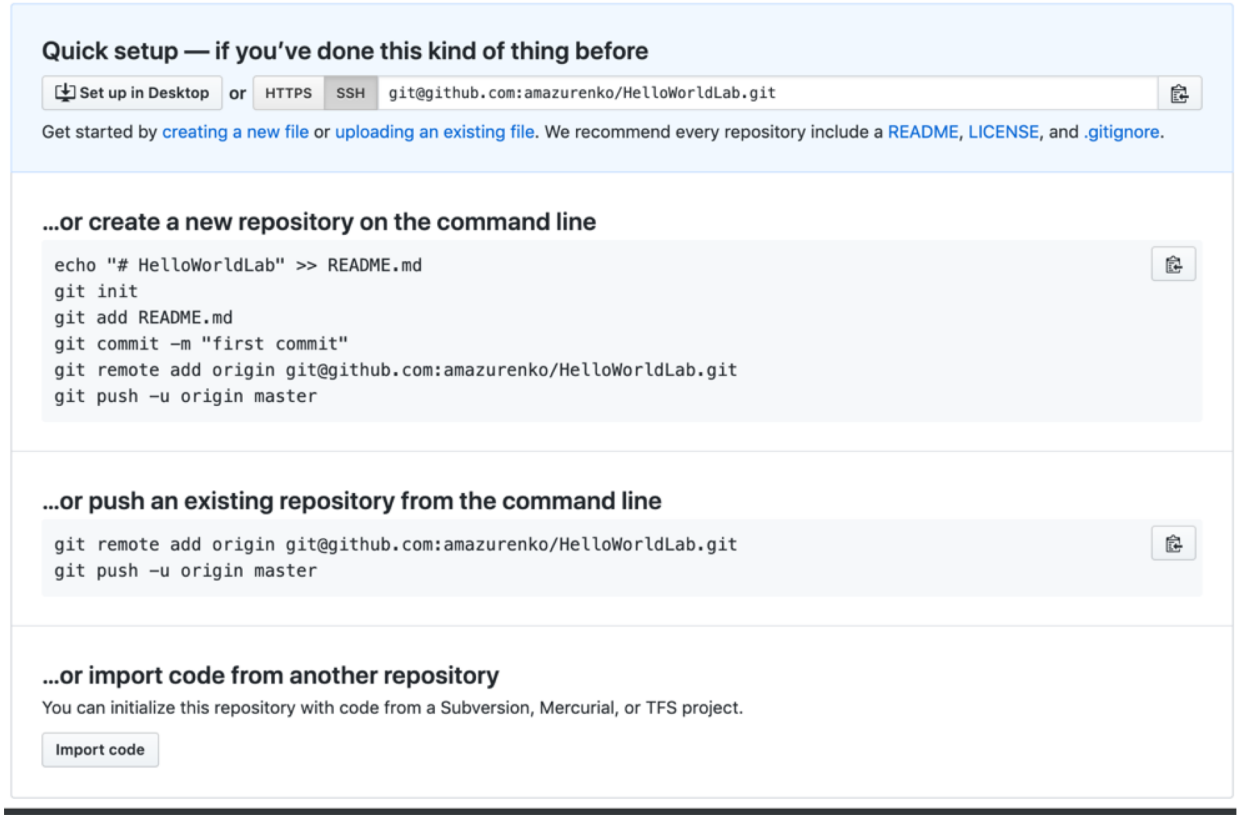
GitHub

GITHUB - OVERVIEW

- https://github.com/
- https://help.github.com/




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 *
amazurenko
/
HelloWorldLab
Great repository names are short and memorable. Need inspiration? How about shiny-enigma?
Description (optional)
Public
Anyone can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.
Add .gitignore: None
Add a license: None
Create repository


Quick setup — if you've done this kind of thing before
Set up in Desktop
or
HTTPS
SSH
git@github.com:amazurenko/HelloWorldLab.git
Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.
...or create a new repository on the command line
echo "# HelloWorldLab" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:amazurenko/HelloWorldLab.git
git push -u origin master
...or push an existing repository from the command line
git remote add origin git@github.com:amazurenko/HelloWorldLab.git
git push -u origin master
...or import code from another repository
You can initialize this repository with code from a Subversion, Mercurial, or TFS project.
Import code