Pro Git

(Everything you need to Know about Git)

Version Control

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

⇛ Git stores everything in its database not by file name but by the hash value of its contents.

Example ⇒ { `24b9da6552252987aa493b52f8696cd6d3b00373` }

⇛ Git Generally Only Adds Data.

⇛ Git has three main states that your files can reside in:

- modified → { Modified means that you have changed the file but have not committed it to your database yet. }
- staged → { Staged means that you have marked a modified file in its current version to go into your next commit snapshot. }
- committed → { Committed means that the data is safely stored in your local database. }

Working Tree

→ The working tree is a single checkout of one version of the project.

→ These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

Staging Area

→ The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

.git directory

→ The Git directory is where Git stores the metadata and object database for your project.

→ It is what is copied when you clone a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

★ First-Time Git Setup

⇒ Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

⇒ These variables can be stored in three different places:

1. /etc/gitconfig
   - → Contains values applied to every user on the system and all their repositories.
   - → If you pass the option --system to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file
   - → Values specific personally to you, the user.
   - → You can make Git read and write to this file specifically by passing the --global option.
3. (.git/config) of whatever repository you're currently using:
   - → Specific to that single repository.
   - → You can force Git to read from and write to this file with the --local option

⇒ Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.

⇒ You can view all of your settings and where they are coming from using:

```
$ git config --list --show-origin
```

★ Your Identity

⇒ The first thing you should do when you install Git is to set your user name and email address.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

⇒ You need to do this only once if you pass the --global option.

⇒ If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you're in that project.

⇒ You can also check what Git thinks a specific key's value is by typing git config <key>.

⇛ Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why.

⇛ In cases like that, you can query Git as to the origin for that value, and it will tell you which configuration file had the final say in setting that value:

```
$ git config --show-origin rerere.autoUpdate
file:/home/johndoe/.gitconfig   false
```

★ Your Editor

```
$ git config --global core.editor emacs
```

★ Getting Help

⇛ If you ever need help while using Git, there are three equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

⇛ For the more concise “help” output with the -h option.

Git Basics

★ Getting a Git Repository

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

1. You can take a local directory that is currently not under version control, and turn it into a Git repository.

   ⇛ Go to your project directory 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.

2. You can clone an existing Git repository from elsewhere.

   ⇛ You clone a repository with git clone <url>.

   ⇛ That creates the directory of the project, 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 want to clone the repository (for ex: libgit2) into a directory named something other than libgit2, you can specify the new directory name as an additional argument:

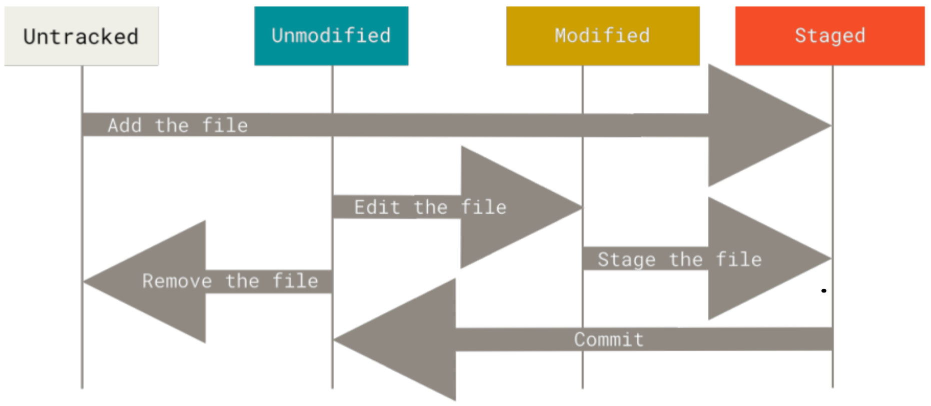
   ```
   $ git clone https://github.com/libgit2/libgit2 mylibgit
   ```

   ⇛ Git has a number of different transfer protocols you can use.

   1. HTTP
   2. SSH

★ Recoding changes to the Repository

⇛ Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.



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

⇛ In order to begin tracking a new file, you use the command git add.

Short Status

```
$ git status -s
```

? ?	New files that aren't tracked.
A	New files that have been added to the staging area.
M	Modified files, but not staget.
M	Modified and staged
MM	Staged and then modified.

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.

```
$ cat .gitignore
*.[oa]
*~
```

$*.[oa] \rightarrow$ {Ignore any files ending in ".o" or ".a"}

$*\sim \rightarrow$ {Ignore all files whose names end with a tilde (~)}

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work, and will be applied recursively throughout the entire working tree.
- You can start patterns with a forward slash (/) to avoid recursivity.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

⇒ Glob patterns are like simplified regular expressions that shells use:

1. An asterisk (*) matches zero or more characters.
2. [abc] matches any character inside the brackets (in this case a, b, or c).
3. (?) matches a single character.
4. [0-9] matches any character between them.
5. You can also use two asterisks to match nested directories; a/**/z would match a/z, a/b/z, a/b/c/z, and so on.

```
# 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
```

⇛ It is also possible to have additional .gitignore files in subdirectories.

↳ The rules in these nested .gitignore files apply only to the files under the directory where they are located.

#Viewing your Staged and Unstaged Changes

git diff   If you want to know exactly what you changed, not just which files were changed.

⇛ To see what you've changed but not yet staged, type git diff with no other arguments.

⇛ If you want to see what you've staged that will go into your next commit, you can use git diff --staged.

⇛ By using -a tag in git commit you can skip git add step.

{ git commit -a -m 'added new benchmarks' }

#Removing files

⇛ To remove a file from Git, you have to remove it from your tracked files and then commit.

⇛ The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

⇛ Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area.

⇛ To do this, use the --cached option.

```
$ git rm --cached README
```

⇛ You can pass files, directories, and file-glob patterns to the git rm command.

#Moving files

⇛ If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

⇛ If you run something like this and look at the status, you'll see that Git considers it a renamed file.

#Viewing the Commit History

⇛ The most basic and powerful tool to do this is the git log command.

⇛ One of the more helpful options is -p or --patch, which shows the difference introduced in each commit.

⇛ You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

★ Undoing Things

⇛ One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message.

⇛ If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:

```
$ git commit --amend
```

⇛ If you've made no changes since your last commit then your snapshot will look exactly the same, and all you'll change is your commit message.

Unstaging a Staged file

$ git reset HEAD <file>...

It's true that git reset can be a dangerous command, especially if you provide the --hard flag.

Unmodifying a Modified file

```
"git checkout -- <file>..."
```

{when file is not staged} but modified

⇛ It's important to understand that git checkout -- <file> is a dangerous command.

- ↳ Any local changes you made to that file are gone — Git just replaced that file with the most recently-committed version.
- ↳ Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.

⇛ Remember, anything that is committed in Git can almost always be recovered.

⇛ Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered.

⇛ However, anything you lose that was never committed is likely never to be seen again.

★ Working with Remotes

⇛ Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

⇛ You can have several of them, each of which generally is either read-only or read/write for you.

⇛ Remote repositories can be on your local machine.

↳ It is entirely possible that you can be working with a "remote" repository that is, in fact, on the same host you are.

⇒ If you've cloned your repository, you should at least see origin — that is the default name Git gives to the server you cloned from.

⇒ To add a new remote Git repository as a shortname you can reference easily, run:

```
git remote add <shortname> <url>
```

Featching and pulling from your remotes

```
$ git fetch <remote>
```

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

→ git fetch command only downloads the data to your local repository, it doesn't automatically merge it with any of your work or modify what you're currently working on.

→ You have to merge it manually into your work when you're ready.

⇒ If your current branch is set up to track a remote branch, you can use the git pull command to automatically fetch and then merge that remote branch into your current branch.

Pushing to your remotes

```
git push <remote> <branch>
```

⇒ If you want to see more information about a particular remote, you can use the:

```
git remote show <remote>
```

Renaming and Removing Remotes

⇒ You can run git remote rename to change a remote's shortname.

⇒ For instance, if you want to rename pb to paul:

```
$ git remote rename pb paul
```

⇒ If you want to remove a remote for some reason

```
$ git remote remove paul
```

★ Tagging

⇒ Like most VCSs, Git has the ability to tag specific points in a repository's history as being important.

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

Listing your tags

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

Creating tags

⇛ Git supports two types of tags:
1. lightweight
2. annotated

{ A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit. }

{ Annotated tags, however, are stored as full objects in the Git database. }

★ Annotated Tags

⇛ The easiest way is to specify -a when you run the tag command:

```
$ git tag -a v1.4 -m "my version 1.4"
```

⇛ You can see the tag data along with the commit that was tagged by using the git show command:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

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

    changed the version number
```

★ Lightweight Tags

⇛ To create a lightweight tag, don't supply any of the -a, -s, or -m options, just provide a tag name:

```
$ git tag v1.4-lw
```

Tagging Later

⇛ You can also tag commits after you've moved past them.

⇛ To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```

Sharing Tags

⇛ By default, the git push command doesn’t transfer tags to remote servers.

⇛ You will have to explicitly push tags to a shared server after you have created them.

⇛ This process is just like sharing remote branches — you can run:

```
git push origin <tagname>
```

⇛ If you have a lot of tags that you want to push up at once, you can also use the --tags option to the git push command.

```
$ git push origin --tags
```

Deleting Tags

⇛ To delete a tag on your local repository, you can use:

```
git tag -d <tagname>
```

⇛ Note that this does not remove the tag from any remote servers.

⇛ There are two common variations for deleting a tag from a remote server.

- The first variation is `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

- The second (and more intuitive) way to delete a remote tag is with:

```
$ git push origin --delete <tagname>
```

Checking out Tags

⇛ If you want to view the versions of files a tag is pointing to, you can do a git checkout of that tag.

```
$ git checkout 2.0.0
```

⇛ Although this puts your repository in “detached HEAD” state, which has some ill side effects.

⇛ In “detached HEAD” state, if you make changes and then create a commit, the tag will stay the same, but your new commit won’t belong to any branch and will be unreachable, except by the exact commit hash.

⇛ Thus, if you need to make changes — say you’re fixing a bug on an older version, for instance — you will generally want to create a branch:

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

Git Branching

⇒ Branching means you diverge from the main line of development and continue to do work without messing with that main line.

Commit Object

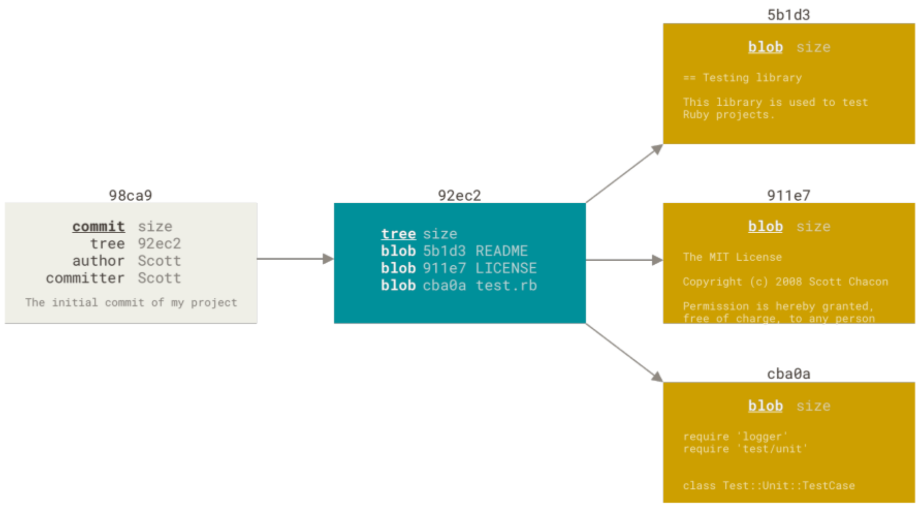
Contains a pointer to the Snapshot of the Content you staged ①

+

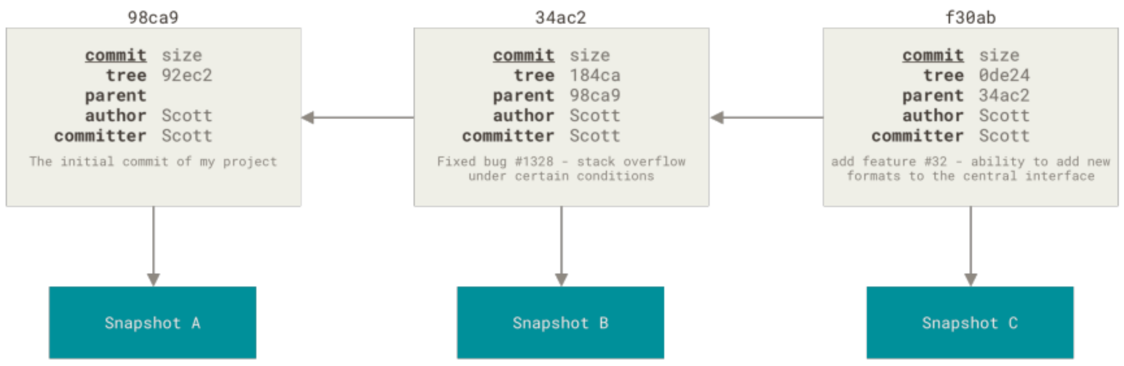
② Metadata

+

③ Pointer to commit or commits that directly Came before this commit



zero parents for the initial commit,
one parent for a normal commit,
and multiple parents for a commit that results from a merge of two or more branches.



⇒ The default branch name in Git is master.

⇒ As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

★ Creating a new branch

⇒ What happens when you create a new branch?

↳ Well, doing so creates a new pointer for you to move around.

⇒ How does Git know what branch you're currently on?

↳ It keeps a special pointer called HEAD.

pointer to the local branch you're currently on

⇛ The git branch command only created a new branch — it didn't switch to that.

★ Switching Branch

⇛ To switch to an existing branch, you run the git checkout command.

⇛ Switching branches changes files in your working directory

⇛ git checkout -b <newbranchname>

└→ Creating a new branch and switching to it at the same time

★ Basic Branching and Merging

⇛ Mearging steps:

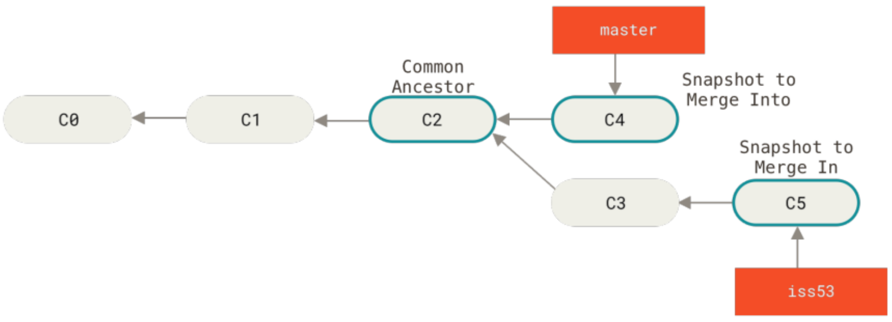
1. Checkout to the branch you want to mearge onto.
2. git merge <branch_to_mearge>

⇛ You can delete branch with the -d option to git branch.

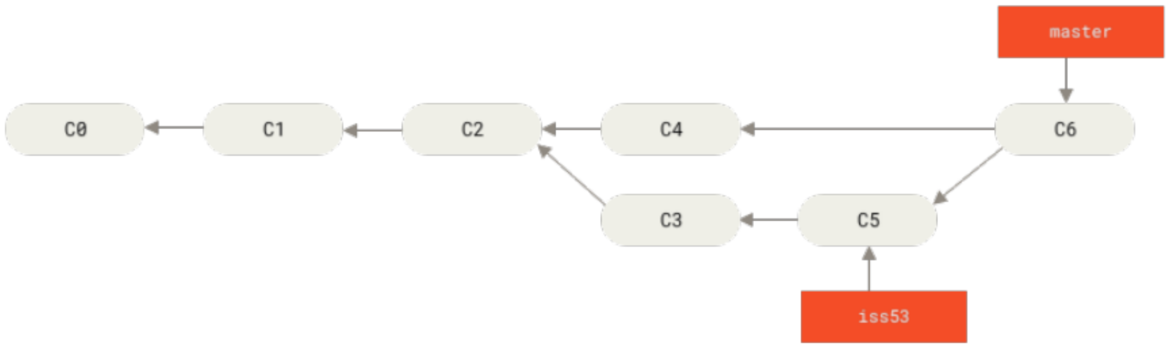
```
$ git branch -d hotfix
```

⇛ In this case, your development history has diverged from some older point.

⇛ Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common.



⇛ Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.



⇛ This is referred to as a merge commit, and is special in that it has more than one parent.

★ Basic Merge Conflicts

- Occasionally, this process doesn’t go smoothly.
- If you changed the same part of the same file differently in the two branches you’re merging, Git won’t be able to merge them cleanly.
- Git hasn’t automatically created a new merge commit.
- It has paused the process while you resolve the conflict.
- If you want to see which files are unmerged at any point after a merge conflict, you can run git status
- Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

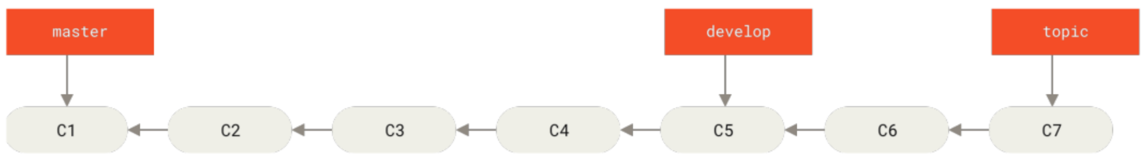
{Example}

- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself.
- After you’ve resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved.
- If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type git commit to finalize the merge commit.

★ Branching Workflows

Long-Running Branches

- Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch.
- They have another parallel branch named develop that they work from.



- The idea is that your branches are at various levels of stability; when they reach a more stable level, they’re merged into the branch above them.

⇛ Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

Topic Branches

⇛ Are useful in projects of any size.

⇛ A topic branch is a short-lived branch that you create and use for a single particular feature or related work.

★ Remote Branches

⇛ Remote references are references (pointers) in your remote repositories, including branches, tags, and so on.

⇛ You can get a full list of remote references explicitly with:

- `git ls-remote <remote>`
- git remote show <remote>

⇛ Remote-tracking branches are references to the state of remote branches.

⇛ They're local references that you can't move;
Git moves them for you whenever you do any network communication,
to make sure they accurately represent the state of the remote repository.

⇛ Remote-tracking branch names take the form <remote>/<branch>.

Deleating remote branches

⇛ You can delete a remote branch using the --delete option to git push.

```
git push origin --delete serverfix
```

{Example}

⇛ Basically all this does is remove the pointer from the server.

★ Rebasing

⇛ In Git, there are two main ways to integrate changes from one branch into another:
1. merge
2. rebase

{ With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch. }

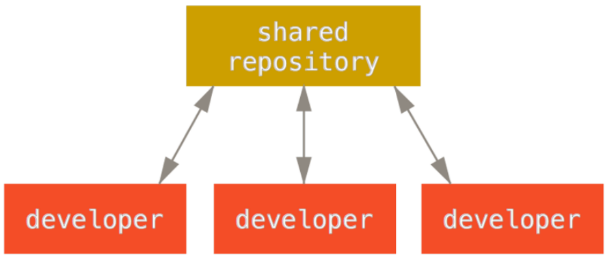
⇛ There is no difference in the end product of the integration, but rebasing makes for a cleaner history.

Distributed Git

★ Centralized Workflow

⇛ In centralized systems, there is generally a single collaboration model.

⇛ One central hub, or repository, can accept code, and everyone synchronizes their work with it.



⇛ This means that if two developers clone from the hub and both make changes,
-> The first developer to push their changes back up can do so with no problems.
-> The second developer must merge in the first one's work before pushing changes up, so as not to overwrite the first developer's changes.

⇛ If you are already comfortable with a centralized workflow in your company or team.
↳ Simply set up a single repository, and give everyone on your team push access
Git won't let users overwrite each other.

★ Integration-Manager Workflow

⇛ Workflow where each developer has write access to their own public repository and read access to everyone else's.

⇛ This scenario often includes a canonical repository that represents the "official" project.

⇛ To contribute to that project, you create your own public clone of the project and push your changes to it.

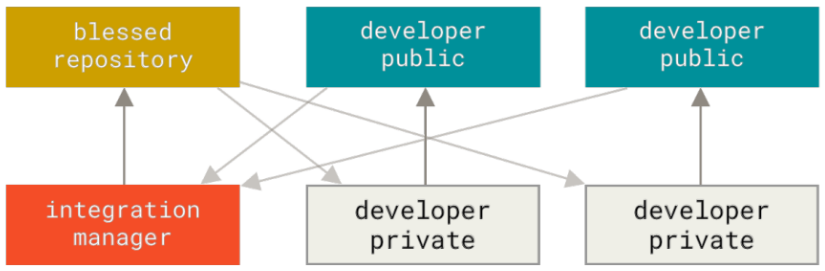
⇛ Then, you can send a request to the maintainer of the main project to pull in your changes.

⇛ The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository.

⇛ The process works as follows:

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor's repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

⇛ This is a very common workflow with hub-based tools like GitHub, where it's easy to fork a project and push your changes into your fork for everyone to see.



★ Dictator and Lieutenants Workflow

⇛ This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators.

↳ One famous example is the Linux kernel

⇛ Various integration managers are in charge of certain parts of the repository; they're called lieutenants.

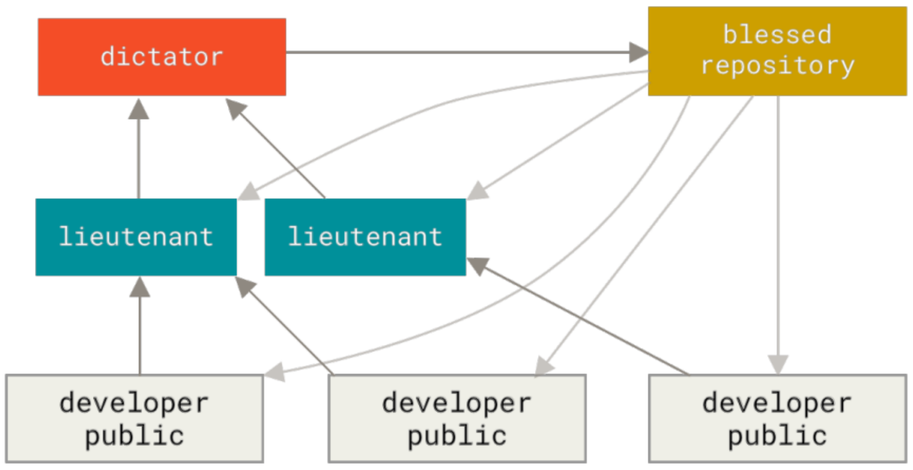
⇛ All the lieutenants have one integration manager known as the benevolent dictator.

⇛ The benevolent dictator pushes from their directory to a reference repository from which all the collaborators need to pull.

⇛ The process works like this:

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the reference repository to which the dictator pushes.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.
4. Finally, the dictator pushes that `master` branch to the reference repository so the other developers can rebase on it.

⇛ This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments.



★ Contributing to a Project

#Commit Guidelines

1. Your submissions should not contain any whitespace errors.

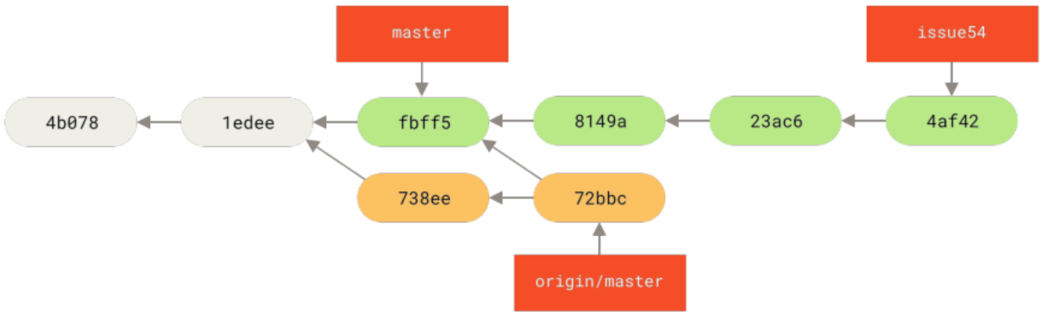
2. Try to make each commit a logically separate changeset.

   { If you can, try to make your changes digestible — don’t code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. }

3. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier.

   { As a general rule, your messages should start with a single line that’s no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed\ explanation. }

   { Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." }



```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   remove invalid default value
```

⇒ The issue54..origin/master syntax is a log filter that asks Git to display only those commits that are on the latter branch (in this case origin/master) that are not on the first branch (in this case issue54).
