Intro to Git

Workshop

Kiorpelidis Platon

University of Athens
ACM UoA Student Chapter

1 Introduction

1.1 Git & Version Control

– *The Problem*: Many people's version-control method of choice is to copy files into another directory. We create copies of the project and name them to *project v2*, *project v3*, etc. 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, delete code that may be useful later – whenever you have the entire history of the project in a single place, you risk losing everything. In addition, it clutters up the workspace and gives vertigo to anyone browsing it.

We also want to collaborate with other developers. We could keep the codebase under a server but what about server failures. If the server goes down for an hour, then during that hour 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.

– *The Solution*: This is where Distributed Version Control Systems step in, like Git. In such systems, 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.

Version Control: It's is a system that records changes to a file or a set of files over time so that you can recall specific versions later.

Distributed Version Control System: Clients fully mirror the remote repository, including it's history. Remote repository is stored in a server. Full copy of the code is present in all the developers' computers.

Popularity of Git (2010 vs 2019):

- 2010: 26,485 repositories (11.3% of total)
- 2019: 913,378 repositories (70% of total)

Server example: GitHub. GitHub is the single largest hosting server for Git repositories with a web front end. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control (users) and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project. A large percentage of all Git repositories are hosted on

GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things. So while it's not a direct part of the Git open source project, there's a good chance that you'll want or need to interact with GitHub at some point while using Git professionally.

1.2 Concept of Git & Early days

Git follows the unix philosophy. The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its creators. The Unix philosophy favors composability as opposed to monolithic design.

In it's early days git consisted of simple, atomic executables. Such programs weren't enough to be useful on their own, but a combination of those, usually in shell scripts, composed more complex functionality like **git add**. Probably it's the reason, after so many developement cycles, still considered a complex and difficult tool.

It was created by Linus Torvalds to help maintain Linux. As more people were sending in patches and the current workflow didn't scale, Linus created a new VCS as he didn't like existing ones. He named it git – the stupid content tracker or the information manager from hell.

1.3 Modern Git

Initially git wasn't developed for the novice user. Nowdays it's a lot friendlier and easier to get into as it comes with a lot of build-in commands, wrappers and complexity abstractions. These commands internally use the old, enhanced, but still atomic executables. There are GUIs developed for the GUI lovers or non-technical people and it's supported with plugins by various text editors and IDEs. These changes and features make for an easier, faster and streamlined collaboration between developers, who follow various local and online workflows.

1.4 Meaning of Git

Copied from the first git commit written by Linus Torvalds.

"git" can mean anything, depending on your mood.

- Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronounciation of "get" may or may not be relevant.

- Stupid. Contemptible and despicable. Simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh*t": when it breaks.

This is a stupid (but extremely fast) directory content manager. It doesn't do a whole lot, but what it does do is track directory contents efficiently.

2 Installing Git

Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

– Linux (Debian, Ubuntu): sudo apt-get install git-all

– Mac: Run git --version. If you don't have it installed already, it will prompt you to install.

– Windows: Download from `https://git-scm.com/download/win`.

3 Working locally

3.1 Initializing Git (git init)

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 cd /home/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. A project that is under Git is called "repository" or "repo". At this point, nothing in your project is tracked yet.

In order to start tracking our changes and as a result our project we need to inform Git about our identity. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

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

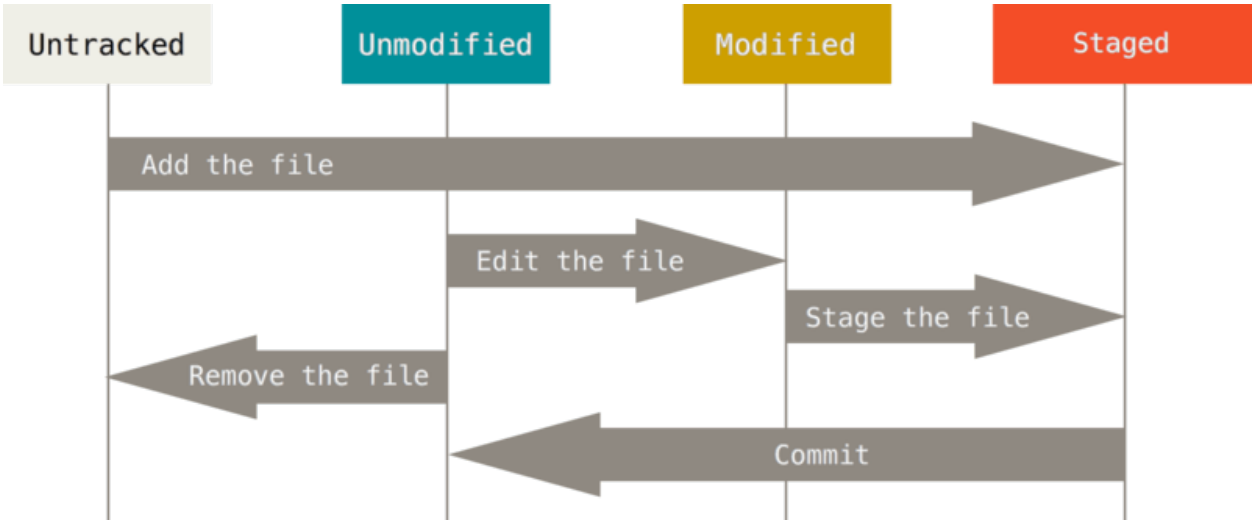
NOTE: -- global sets your name and email across Git projects. If you want to have a different identity for different projects run the above instructions without -- global flag. You can check your settings with git config -- list.

```
commit 2973c7bb0d774dcb9b817c0d120ae1abeb3011c1
Author: Plato <otalpster@gmail.com>
Date:   Tue Nov 19 16:40:54 2019 +0200

    testfile.c: initial commit
```

– *Exercise*: Open your laptops! Open a terminal and create a directory. Inside the new directory run git init. Then run ls -la and notice the hidden directory .git.

3.2 The three stages



Each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first create a file it will be untracked because Git does not track it yet.

As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

3.3 Checking the status of your file (git status)

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

– *Exercise*: Run git status. Create a new file touch testfile. Run git status again. The new file should appear as untracked.

3.4 Staging changes (git add)

git add moves changes to the staging area. Changes to Git means anything it does not know, new files or modified ones. The staging area is where we are cooking the contents of our next snapshot.

– *Exercise*: Let's track our new file. Run git add. See that our new file is in the staging area with git status. Now our untracked file is cooking for the next snapshot! When Git records the current staging area and creates a new snapshot our untracked file will become tracked. Until then it's just staged.

– *Exercise*: Open your editor of choice and change the testfile. Add some random letters. Run git status. Notice that the testfile is listed as both staged and unstaged.

How is this possible? It turns out that Git stages a file exactly as it is when you run git add. If you record a new snapshot now, the version of testfile as it was when you last ran git add is how it will go into the snapshot, not the version of the file as it looks in your working directory. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file.

3.5 Viewing your staged and unstaged changes (git diff)

Sometimes, git status command is too vague. You want to know exactly what you changed, not just which files were changed. To achive this you use the git diff command. You most often use it to answer these two questions: What have you changed but not yet staged? What have you staged that you are about to commit? git status answer these questions by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were.

To see changes you've made that you haven't yet staged, type git diff. That command compares what is in your working directory with what is in your staging area.

To see what you've staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit.

– *Exercise*: View your changes. Run git diff. Run git diff --cached. Notice the + and - characters. What do they mean? +, - describes the added and removed lines respectively.

It's important to note that git diff by itself doesn't show all changes made since your last commit – only changes that are still unstaged. If you've staged all of your changes, git diff will give you no output. You will need to run git diff and git diff --staged to show all changes made since your last commit, staged and unstaged.

git status can reproduce the above behaviour by running internally and formatting the output of git diff and git diff --staged with the flag git status -v/-vv. The -v flag, in addition to the output of git status, will also show the actual changes that are staged to be committed, like git diff --cached. The -vv flag, in addition to the output of git status -v, will also show the changes that have not yet been staged, like git diff.

– *Exercise*: View your changes. But this time run git status -v. Run git status -vv. See the difference and the similarity with git diff/git diff --cached?

3.6 Snapshoting/Committing your changes (git commit)

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them – won't go into this commit. They will stay as modified files on your disk. So you're ready to commit your changes. The simplest way to commit is to type git commit.

Each commit consists of the **commit message** and a **body**. It's a common practise to write commit messages with up to 50 characters. Commit messages are used for a brief but descriptive explanation about what changes you are committing. The body should provide a meaningful commit message, for example:

- Explains the problem the change tries to solve, i.e. what is wrong with the current code without the change.
- Justifies the way the change solves the problem, i.e. why the result with the change is better.
- Alternate solutions considered but discarded, if any.

If your description starts to get too long, that's a sign that you probably need to split up your commit to finer grained pieces. Descriptions that summarize the point in the subject well, and describe the motivation for the change, the approach taken by the change, and if relevant how this differs substantially from the prior version, are all good things to have.

– *Exercise*: Commit your changes. Run git commit. Doing so will launch the default editor with the output of git status commented out. The first line is the commit message. Leave a blank line and write a short description.

– *Exercise*: Stage your changes. Run git add. Now commit them. Sometimes you forget what you are committing and need even more explicit reminder than the commented out git status. Run git commit -v. Doing so also puts the diff of your change in the editor so you can see exactly what changes you're committing. Save and exit the editor to create your commit.

Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
```

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

3.7 Viewing the commit history (git log)

After you have created several commits, or if you collaborate with others and they made changes (more later), 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.

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

A huge number and variety of options to the git log command are available to show you exactly what you're looking for. Here, we'll show you some of the most useful.

– *Exercise*: Run git log --one-line. One line per commit.

– *Exercise*: Run git log -n. Show the last n commits.

– *Exercise*: Run git log -p. Show the changes introduced.

– *Exercise*: Run git log --stat. Abbreviated commit stats.

3.8 Undoing staged and modified changes

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

3.8.1 Unstaging a Staged File

You made a mistake and ran git add against a file which you don't want to commit yet. How can you unstage it? The git status command reminds you.

– *Exercise*: Make a very simple change. Run git add. Now run git status. Run git reset HEAD testfile to unstage. The command is a bit strange, but it works. The testfile file is modified but once again unstaged.

NOTE: git reset can be a dangerous command. However, in the scenario described above, the file in your working directory wasn't touched, so it's relatively safe.

3.8.2 Unmodifying a Modified File

What if you realize that you don't want to keep the changes you made to a file? How can you easily unmodify it – revert it back to what it looked like when you last committed? Luckily, git status tells you how to do that, too.

– *Exercise*: Run git status. Run git checkout -- testfile. You can see that the changes have been reverted.

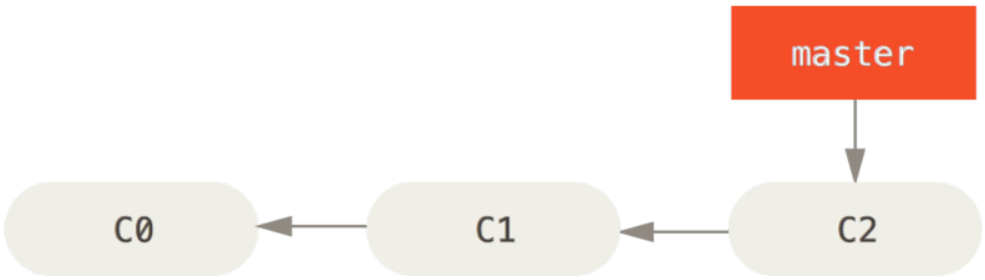
NOTE: It's important to understand that git checkout -- <file> is a dangerous command. Any local changes you made to that file are gone – Git just replaced that file with the most recently-committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.

Know that anything that is committed in Git can almost always be recovered. Even commits that were deleted or commits that were overwritten/modified can be recovered. However, anything you lose that was never committed is likely never to be seen again.

3.9 Branching

Branching means you diverge from the main line of development and continue to do work without messing with that main line. Sometimes you want to experiment with an idea, implement a new feature or fix a bug, in a seperate "container", without messing or changing

the main line, stable "master" branch, making it unstable – with unfinished, bug-ridden – work in progress changes. Maybe you were working on a new feature and an critical bug needs to be fixed immediately. You need to somehow save your work and fix that bug.



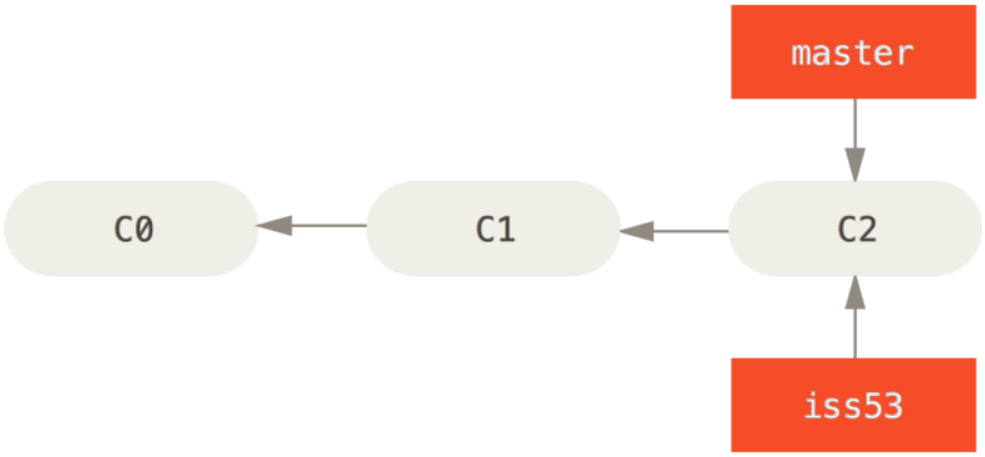
A branch is simply a movable pointer to a commit. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don't bother to change it.

Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

– *Exercise*: List the existing branches. Run git branch. Notice that only master exists.

3.9.1 Create a branch



Let's create a new branch. Creating a branch "testbranch" means we create a new pointer, named "testbranch" for us to move around.

– *Exercise*: Create a new branch. Run git branch testbranch. This creates a new pointer to the same commit you're currently on. git branch only created a new branch – it didn't switch to that branch. Run git branch again. Notice the new branch and the branch you are on which is indicated by the asterisk.

– *Exercise*: Where are the branch pointers pointing? Run git log. You can see the "master" and "testing" branches that are right there next to the last commit.

3.9.2 Working on branches and diverging history

Let's switch to our new branch and make a change. To switch to an existing branch, you run the git checkout command.

– *Exercise*: Switch to the new branch. Run git checkout testbranch. Run git branch. Notice the asterisk is next to testbranch now.

– *Exercise*: Open the testfile file and make a tiny change. Run git status and verify the file indeed changed. You may want to see the actual changes in which case run git diff. Stage the changes with git add and commit them with git commit -m "Tiny change". You can pick any commit message you think is correct. Run git log and notice that testbranch moved forward and master did not.

Let's change to master branch and also make a change there.

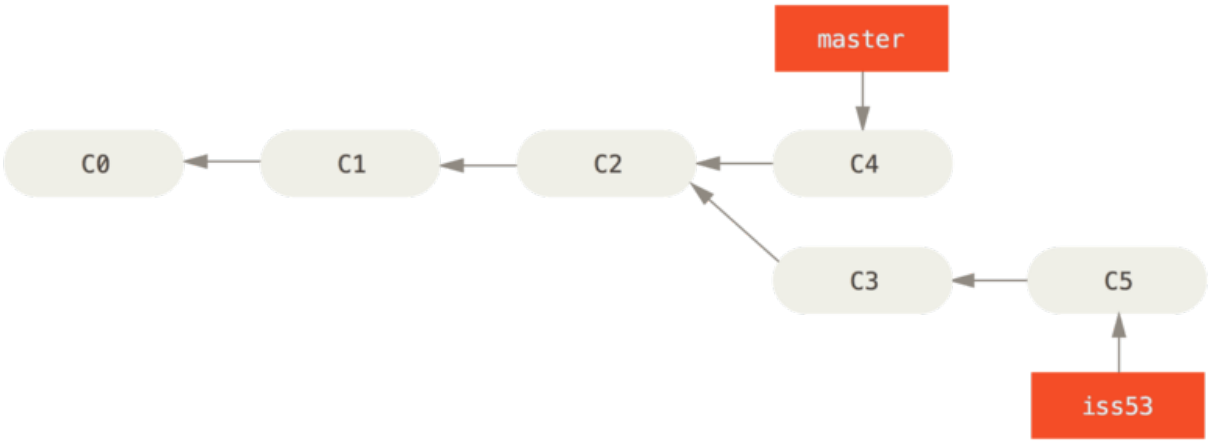
– *Exercise*: Change back to master branch. Run git checkout master.

Notice the files in your working directory reverted back to the snapshot that master branch points to. If Git cannot do it cleanly (you have uncommitted changes that conflict), it will not let you switch at all. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your testbranch branch so you can go in a different direction.

– *Exercise*: Change testfile file. Run git add and git commit -m.

Now your project history has diverged. You created and switched to a branch, did some work on it, and then switched back to master branch and did other work. Both of those changes are isolated in separate branches. You can switch back and forth between the branches.

– *Exercise*: We can easily see the diverged history with git log. Run git log --oneline --graph --all.



3.9.3 Merging branches

You finished a piece of work, tested it and you want to merge/unify it back into the main line of development. You can do that using git merge. Let's do it.

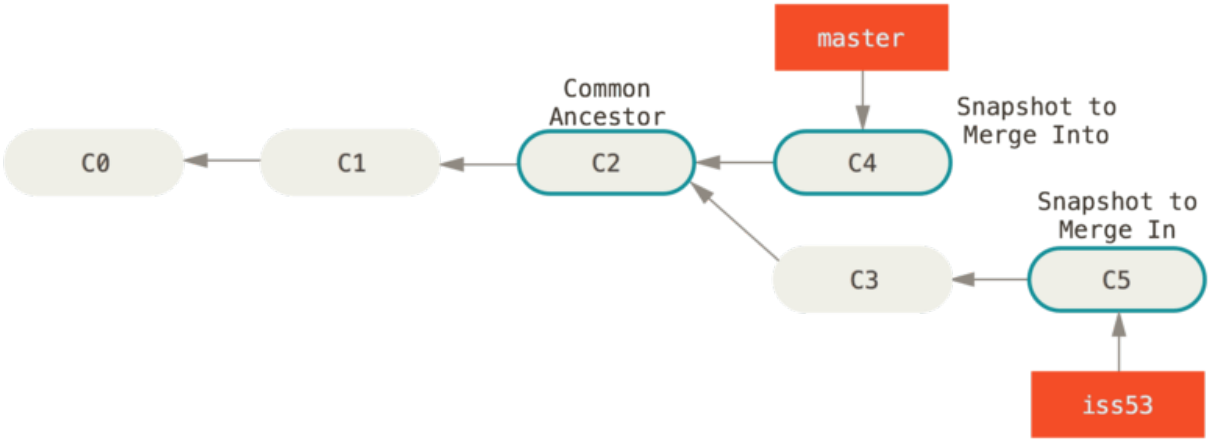
– *Exercise*: Switch to the branch you want to merge your feature in. In our example we are already in master branch and we want to merge testbranch. Run git merge testbranch.

Git has various algorithms/strategies to merge branches together. git merge will automatically select a merge strategy unless explicitly specified. There are explicit and implicit merges. We will cover fast-forward and recursive methods. Fast-forward is an implicit merge. We can perform a fast-forward merge if the branch we are merging into has not diverged. In such case Git simplifies things by moving the pointer forward because there is no divergent work to merge together. Recursive merge on the other hand is an explicit merge. The explicit part is that they create a new m̈ergec̈ommit. Recursive is the default merge strategy when merging one branch. Git will attempt to find a common ancestor between the two branches. Once git finds a common ancestor it will create a new m̈erge commitẗhat combines the changes of the two branches. Technically, a merge commit is a regular commit which just happens to have two parent commits.

In this case, your development history has diverged from some older point. Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work as it cannot fast-forward.

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. If your change in branch master modified the same part of a file as the testbranch branch,

you'll get a merge conflict. In other cases, If you didn't change the same part of the file or the branch you are merging into does not have any changes after you branched out, it will merge cleanly.

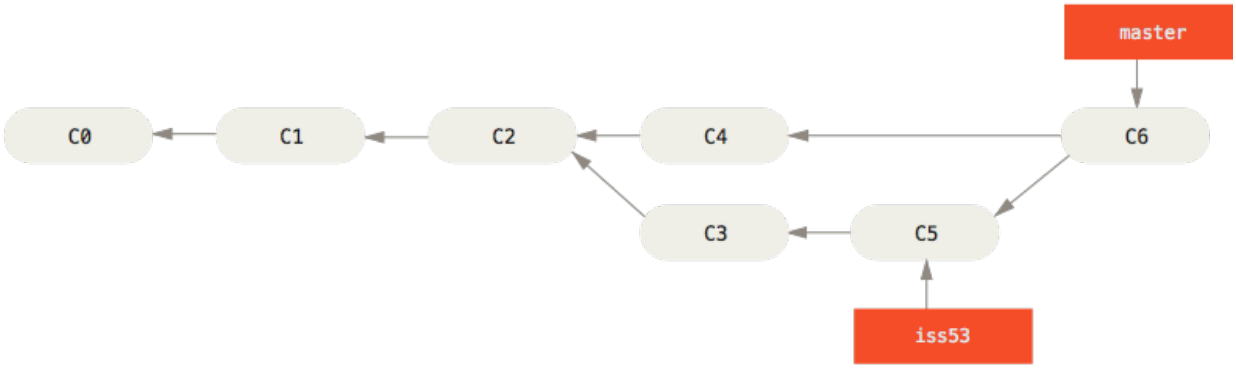


Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Merge conflicts appear with a block that contains `<<<<<<<`, `=======` and `>>>>>>>`. The top part of that block, everything above `=======`, is the branch you are merging into and everything bellow is the branch you are merging in. 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. Staging the file marks it as resolved in Git.

At this point, after running `git merge` you should have a merge conflict. Changes from the testbranch conflict with changes from master branch. 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`.

– *Exercise*: Open testfile file and resolve the conflict. Notice the conflict block with the markers. Solve the conflict and stage the file. Create the merge commit with `git commit`.

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.



3.9.4 Deleting branches

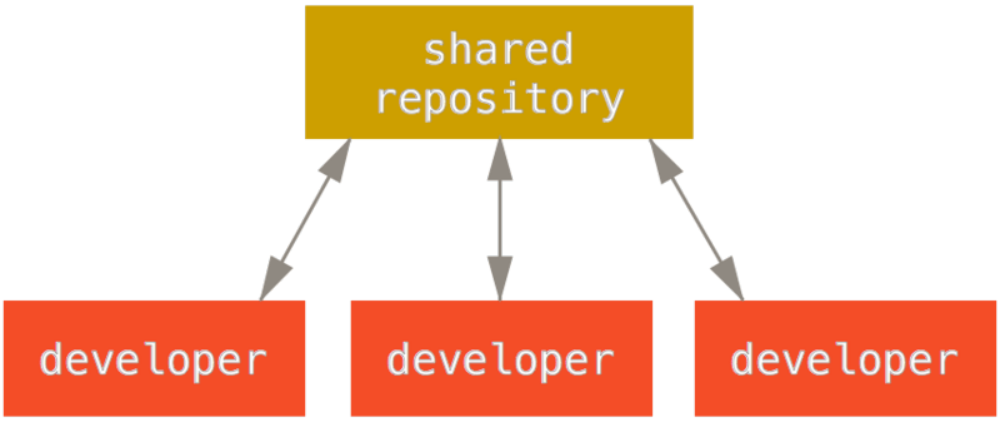
After you implement or fix a bug you no have no further need for the branch. You can delete it with git branch -d <branchname>. This command deletes branches that have been merged into the branch you are currently in. If the branch you are trying to delete contains work that isn't merged in yet, trying to delete it will fail. If you really want to delete such branch and lose that work, you can force it with -D flag, git branch -D <branchname>.

– *Exercise*: Delete branch testbranch. Run git branch -d testbranch. Run git branch and notice the branch no longer exists.

4 Distributed workflows – Git & GitHub

In Git, every developer is potentially both a node and a hub; that is, every developer can both contribute code to other repositories and maintain a public repository on which others can base their work and which they can contribute to. This presents a vast range of workflow possibilities for your project and/or your team. We'll cover a few common paradigms that take advantage of this flexibility.

4.1 Centralized Workflow



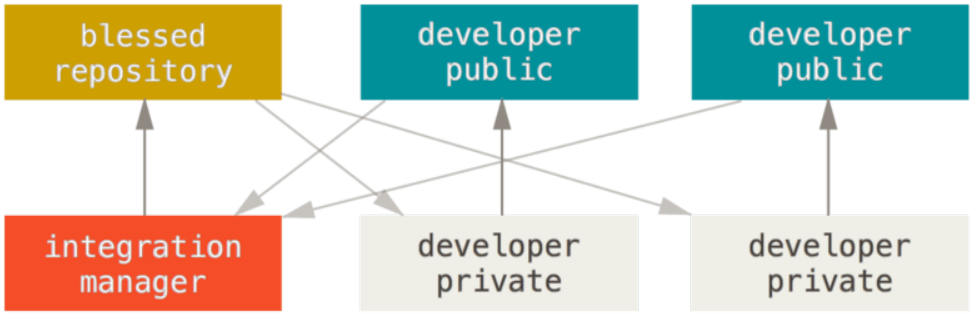
One central hub, or repository, can accept code, and everyone synchronizes their work with it. A number of developers are nodes – consumers of that hub – and synchronize with that centralized location.

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 tries to push their changes, but the server rejects them. They are told that they're trying to push non-fast-forward changes and that they won't be able to do so until they pull the changes and solve possible merge conflicts when pulling the first developer's changes.

To achieve something like that, simply set up a single repository on a server (probably GitHub), and give everyone on your team push access; Git won't let users overwrite each other.

4.2 Integration-Manager Workflow

Because Git allows you to have multiple remote repositories, it's possible to have a 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.



This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes – each party can work at their own pace.

4.3 Working with remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes, manage various remote branches, and more.

GitHub can serve as a host for remote repositories. It's the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things.

– *Exercise*: Create a repository on your GitHub account. Log in and click up right. Click on "New repository". Give it a name and click "Create repository". Git helps you get started with your new empty repository.

4.3.1 Adding remotes

We reference remotes by a shortname as it's easier to remember. To add a new remote repository as a shortname you can reference easily run git remote add <shortname> <url>.

– *Exercise*: Link the existing project with the new remote on GitHub. Run git remote add origin https://github.com/<yourusername>/<yourrepo>.git

4.3.2 Listing remotes

To see which remote servers you have configured, you can run the git remote command. It lists the shortnames of each remote you've specified. You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote.

– *Exercise*: List the current remotes. Run git remote. It should list the origin remote we previously configured. Run git remote -v to list it's URL as well.

4.3.3 Pushing to remotes

Let's upload our changes. What changes? Changes that are in which branch? master branch? In order to upload them we have to upload our branch. The command for this is git push <remote> <branch-shortname>.

– *Exercise*: Upload our master branch to GitHub remote repo. Run git push origin test-branch. Go to the GitHub repo and see how it updated and now shows the actual project instead of the "Quick setup".

4.3.4 Pulling from remotes

As other people work on our project and push their changes – you added them as collaborators, so they have write rights – we want to download them, pull them as we call it. However your repo is new and you just started out, so you don't have other people pushing changes. So let's collaborate together while we take a look at a distributed workflow using GitHub.

4.4 GitHub

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you're collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the Topic Branches we covered in Branching.

Here's how it generally works:

- Fork the project
- Create a topic branch from master.
- Make some commits to improve the project.

- Push this branch to your GitHub project.
- Open a Pull Request on GitHub.
- Discuss, and optionally continue committing.
- The project owner merges or closes the Pull Request.
- Sync the updated master back to your fork.

This is basically an implementation of Integration Manager workflow by GitHub.

4.4.1 Forking projects

When you "fork" a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, your user, and you can push to it. Then you contribute your changes back to the original repository.

– *Exercise*: Open GitHub and fork the example repository at `https://github.com/otalpster/test-repo.git` by clicking fork at the top right. After a few seconds, you'll be taken to your new project, with your own writeable copy of the code under your user.

4.4.2 Cloning a repository

There are typically two ways to obtain a Git repository. We already covered the first way with git init, taking a local directory that is currently not under version control, and turn it into a Git repository. The second way is downloading an existing Git repository from somewhere else. We can do that with git clone <url>.

– *Exercise*: Clone the forked repository locally. Run git clone https://github.com/<yourusername>/test-repo.git.

This creates a directory named test-repo, 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 test-repo directory that was just created, you'll see the project files in there, ready to be worked on or used.

4.4.3 Pull requests

We come up with an awesome new feature for the Linux kernel. We forked and cloned our fork locally. We create a topic branch, make and commit the changes and push them up to GitHub.

If we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and presents us with a big green button to check out our changes and open a Pull Request to the original project.

If we click that green button, we'll see a screen that asks us to give our Pull Request a title and description, as well as a list of the commits in our topic branch that are "ahead" of the master branch and will be merged, if the project owner accepts. It is almost always worthwhile to put some effort into this, since a good description helps the owner of the original project determine what you were trying to do, whether your proposed changes are correct, and whether accepting the changes would improve the original project.

When you hit the Create pull request button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

– *Exercise*: Create a new branch and make a change. Push it to your fork and open a Pull Request to the "official" repository. If the project owner accepts and merges a Pull Request, other Pull Requests that affect the same code section will contain merge conflicts.

NOTE: It's important to note that you can also open a Pull Request between two branches in the same repository. If you're working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the master branch of that same project to initiate the code review and discussion process. No forking necessary.

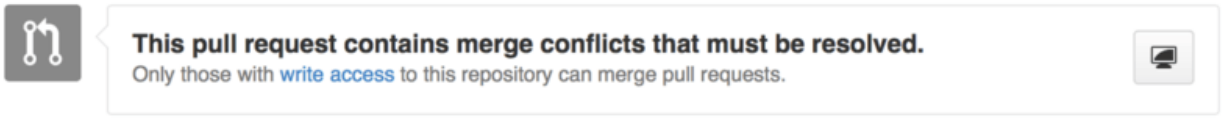
4.4.4 Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Usually this conversation takes place over email, on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines. Once the maintainer makes this comment, the person who opened the Pull Request will get a notification.

Now the contributor can see what they need to do in order to get their change accepted. Luckily this is very straightforward. Simply commit to the topic branch again and push, which will automatically update the Pull Request.

The other thing you'll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a merge. If there are merge conflicts it will prompt you to fix them through the website.

4.4.5 Sync your fork



You forked a project and you started working on it. As you work other contributors/colleagues send their requests to the maintainer to pull in their work. Their work gets pulled and your fork on GitHub becomes out of date. On top of that your changes probably don't merge cleanly, if you opened a pull request on GitHub it will show in red, because of merge conflicts with the updated remote. You'll want to update your fork, fix your branch so it merges cleanly and the maintainer doesn't have to do extra work merging it theirself.

If you want to make your Pull Request mergeable, you would add the original repository as a new remote, pull from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on. Then, to update your fork push the main branch to your remote.

– *Exercise*: Sync your fork. Add the "official" repository as a remote named "upstream". Run git remote add https://github.com/otalpster/test-repo. Pull the main branch of that remote. Run git pull upstream master. Merge the main branch into your topic branch, checkout topic branch and run git merge master. Fix the conflicts that occurred. Make the merge commit. Push back up to the same topic branch. Run git push. Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges cleanly.

Thank you!

You should feel fairly comfortable contributing to a project in Git as well as maintaining your own project or integrating other users' contributions. Congratulations on being an effective Git user!