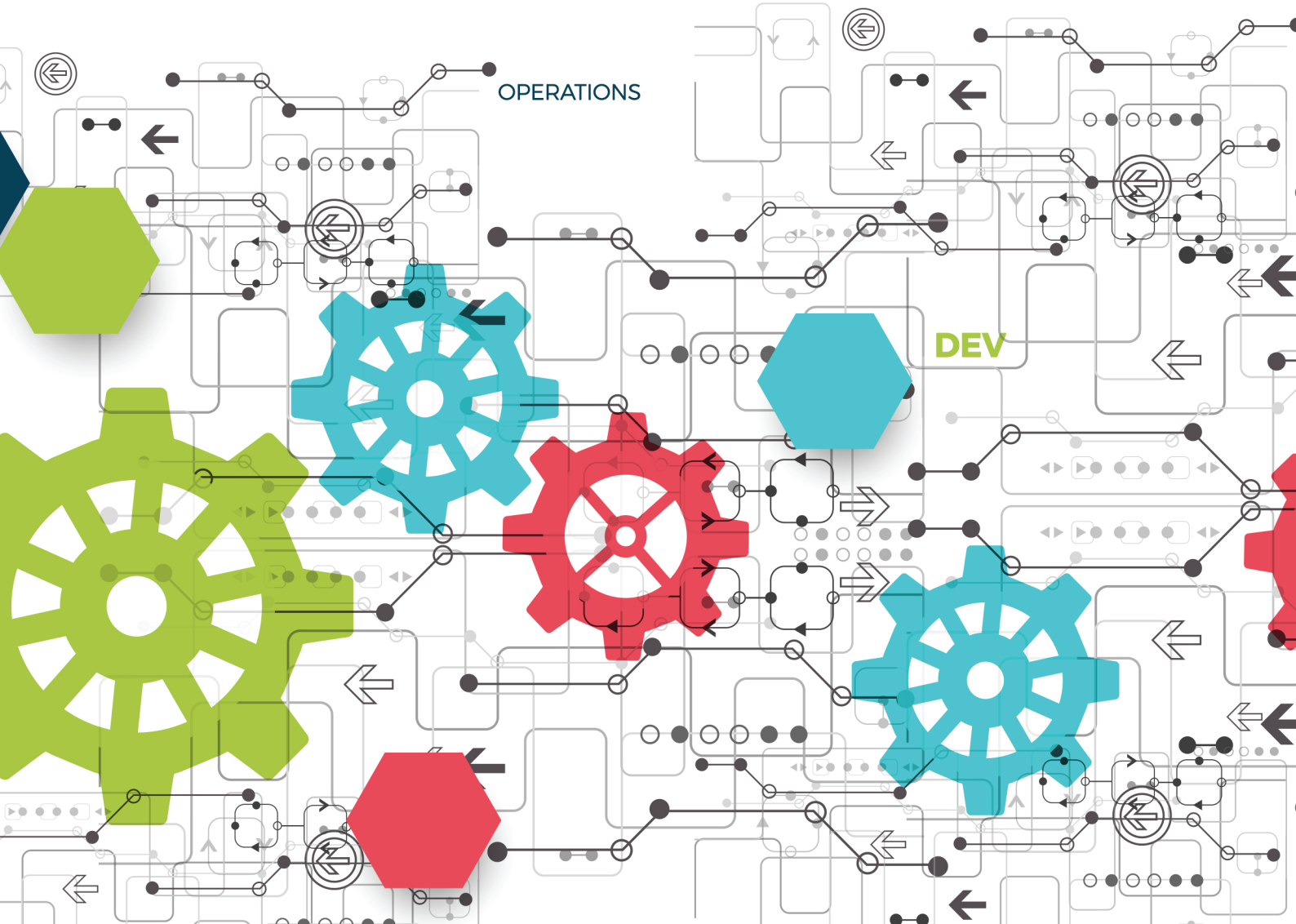
B.Tech Computer Science and Engineering in DevOps

Source Code Management

Additional Appendix
GIT



Git

Introduction

Git is a distributed version control system similar to Mercurial. Both Git and Mercurial were created with a goal of replacing BitKeeper, the version control system used by the Linux kernel group then. Git was created almost single-handedly by Linus Torvalds, the creator of Linux. Linux group had the following goals in mind while developing the alternative version control system.

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its creation in 2005, Git has been the most popular tool for source code management and version control. The truly distributed nature of Git has attracted many developers, who themselves use this product for their individual source code management. Git works well with most of the operating systems and IDEs.

Important features of Git

The three important features of Git are performance, security and flexibility.

Performance

Git exceeds many of its counterparts in performance. Git has been optimized for performance in terms of committing changes, branching, merging and comparing past versions. Git doesn’t look at the name of the files while determining the storage and the version history of the file. Instead, Git looks at the content of the files. Git uses a combination of delta encoding (storing content differences), compression and explicitly stores directory contents and version metadata objects. Many of the version control systems store information as a list of file-based changes (delta-based version control). On the other hand, Git handles data more like a series of snapshots of a file system. During each commit, Git

takes a snapshot of the files in that project and stores a reference to that snapshot. Most importantly, if files are not changed, that file is not saved again.

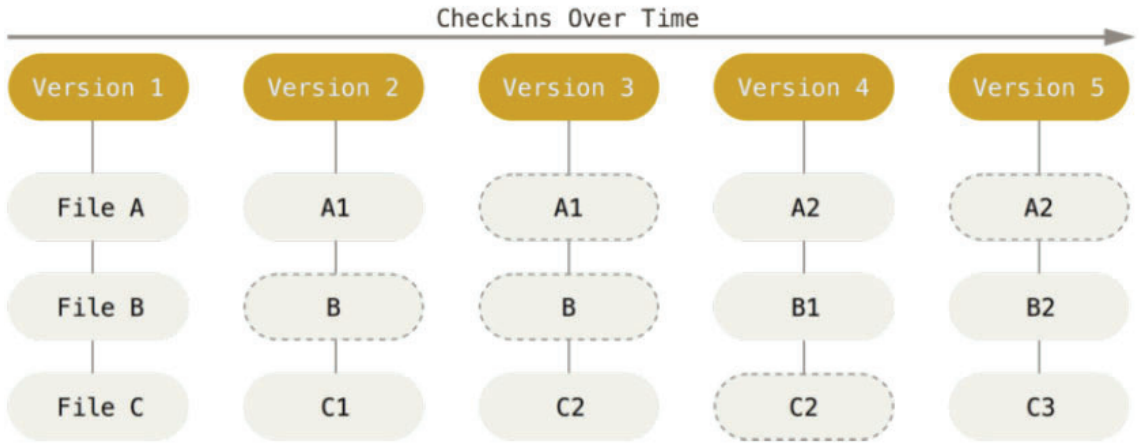


Image source: Pro Git

Security

One of the major goals behind the creation of Git is integrity. Git uses a hashing algorithm called SHA1 to secure the file contents, directories, versions, tags and commits. Everything in Git is check-summed before it is saved, and is then referred to by the checksum. SHA-1 hash is a 40-character string composed of hexadecimal characters (0–9 and a–f) and is calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this: *24b9da6552252987aa493b52f8696cd6d3b00373*. This makes the history completely traceable and protects the code against accidental and malicious damage. Most other version control systems lack this feature and can lead to serious information security vulnerability.

Flexibility

Git is flexible in a number of aspects. It supports multiple nonlinear workflows. Git is efficient in handling both small- and large-scale projects.

Basics of Git

Three Stages of a File

Within Git, our files travel through three different stages as follows:

- Committed- data is saved in the local database.
- Modified – changes are made to the file, but the file is not committed to the database yet.
- Staged – a modified file has been marked in its current version to go into the next commit snapshot.

With this, it is important to know three important concepts of a Git project.

- The Git directory – the location where Git stores the metadata and object database for our project. It is this Git directory that is copied when a repository is cloned from another computer.
- The working tree - 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, which we can use or modify.
- The staging area – present in the Git directory. The staging area stores information about what will go into the next commit. It is technically named as “index”, but commonly referred to as “staging area”.

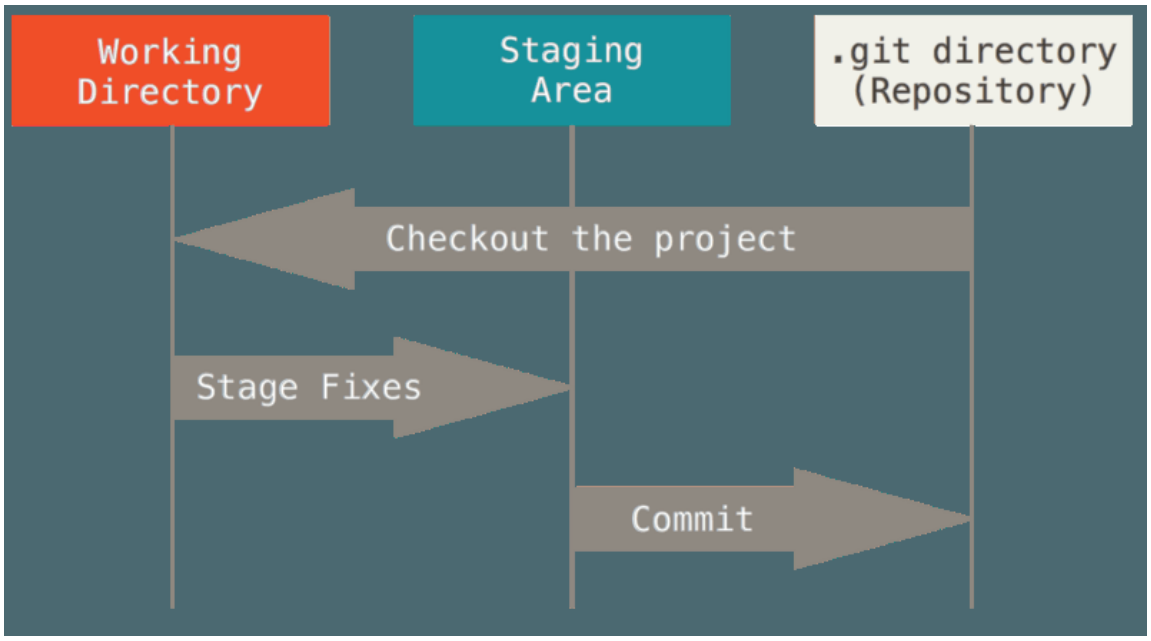


Image source: Pro Git

Basic Git Workflow

- File in the working directory is first modified.

- Selected changes that are to be there in the commit (only) are moved to the staging area.
- Staged changes are then committed, and the commit process takes the files in the staging area and stores the snapshot permanently to the Git directory. If something is there in the Git directory, it is considered to be committed.
- A file that has been modified and added to the staging area is considered staged.
- A file that has been changed after being checked out from the Git repository, but not staged is considered modified.

Installing Git

Git for Linux

1. Debian/Ubuntu systems, Git packages are available via the package manager apt.

From the shell, Git can be installed using apt-get.

$ sudo apt-get update

$ sudo apt-get install git

Successful installation of Git can be verified using the command git --version.

$ git --version

Once Git is installed, Git username and password can be configured using the git config command.

$ git config --global user.name “user1”

$ git config --global user.email “user1@abc.com”

This username and email will be used by Git for each commit.

2. For Fedora users

Git packages are available in both yum and dnf. From the shell, type one of the following commands:

$ sudo yum install git

Or

$ sudo dnf install git

Installation can be verified and username and email can be configured in the same way, using the same commands as described above for Debian systems.

Git for Windows

1. Git package for Windows can be installed from the following link: https://git-for-windows.github.io/.
2. On starting the installer, the Git Setup wizard will pop, using which you can complete the setup.
3. From the command prompt, run the commands for configuring the username and email for Git.

$ git config --global user.name “user1”
$ git config --global user.email “user1@abc.com”

You can also use Git Bash if during installation you had selected not to use Git from the Windows Command prompt.

Checking the Git settings

The command git config --list can be used to check the configuration settings of Git.

git config --list

user.name = user1

user.email = userone@abc.com

color.status = auto

color.branch = auto

color.interactive = auto

color.diff = auto

...

Git Help

Whenever we need to get help in Git, there are two ways to get the comprehensive manual page (manpage) help for any of the Git commands.

$ git help <verb>

$ man git-<verb>

For example, we can get the help for git config command by running

$ git help config

If we do not need the complete man-page help, but just a refresher on available options for any Git command, we can use -h or --help options.

$ git add -h

usage: git add [<options>] [--] <pathspec>...

-n, --dry-run dry run

-v, --verbose be verbose

-i, --interactive interactive picking

-p, --patch select hunks interactively

-e, --edit edit current diff and apply

-f, --force allow adding otherwise ignored files

-u, --update update tracked files

-N, --intent-to-add record only the fact that the path will be added later

-A, --all add changes from all tracked and untracked files

--ignore-removal ignore paths removed in the working tree (same as --no-all)

--refresh don't add, only refresh the index

--ignore-errors just skip files which cannot be added because of errors

--ignore-missing check if - even missing - files are ignored in dry run

--chmod <(+/-)x> override the executable bit of the listed files

Working with a Git Repository

A Git repository (repo, in short) is a directory where the versions of the project is stored, which can be accessed as needed.

Initializing a New Repository

A new repository can be initialized using the git init command. This is a one-time command used during the initial setup of a new repo. This command will create a new .git subdirectory in the current working directory. This will also create a new master branch. Let's say we have our source code in our local computer in a directory. First go to the project directory using cd and execute the git init command.

cd /path/to/your/existing/code

git init

We can point git init to an existing project directory using the same initialization command as above.

git init <project directory>

Cloning an existing repository using git clone

If the project has already been set up in a central repository, the clone command is commonly used to get a local development copy.

git clone <repo url>

git clone command is used to create a copy of the remote repository. We pass the URL of the repository to the git clone command. The url of the project to be cloned can be from source code repository located in GitHub, BitBucket, etc.

Saving the changes to a repository using git add and git commit

We now have a repository initialized or cloned, we can commit file version changes to it. The steps as follows:

- Change directories to /path/to/project
- Create a new file Testfile.txt with contents ~"test content"~
- git add Testfile.txt to the repository staging area
- Create a new commit with a message describing the work was done in the commit

cd /path/to/project

echo "test content" >> Testfile.txt

git add Testfile.txt

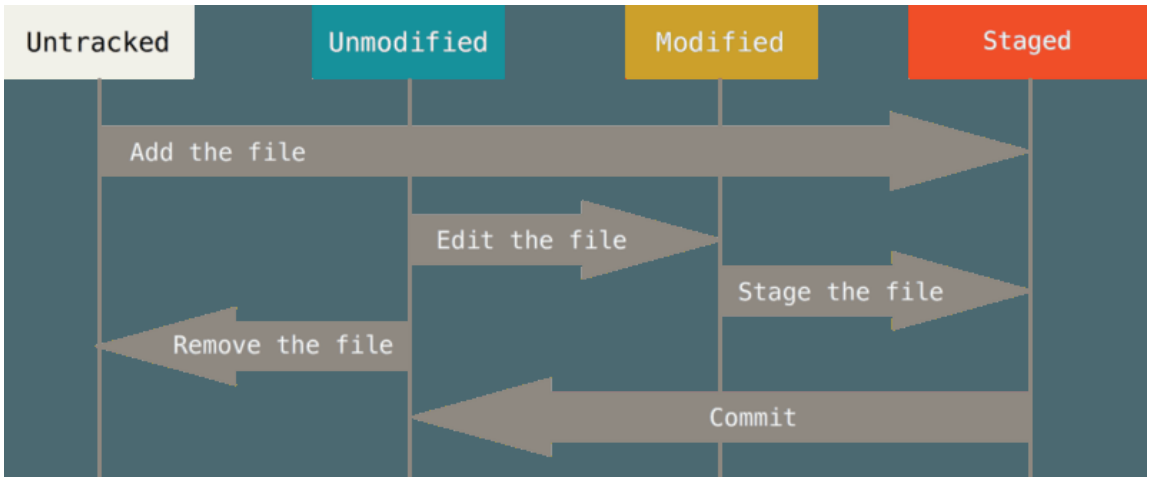
git commit -m "added Testfile.txt to the repo"

After executing the above command, Testfile.txt will get added to the repository's history and will track the future work done to the file. We also have done an initial commit to this repository.

Recording the Changes to the Repository

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 clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.



The git status command is used to determine which files are in which state. After cloning a repository, if you run the status command directly, you will get:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

The above message means that the working directory is clean and none of the tracked files are modified. The command also tells you the branch that you're in and master also means that it has not diverged from the same branch on the server.

For example, if we're adding a new file README. If the file doesn't exist before and if we run git status command, we'll get something like this:

$ echo 'My Project' > README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

This means that the new README file is untracked, because it is under 'Untracked files'. An untracked file according it Git, is the one that is not there in the previous commit. Unless explicitly asked to do so, Git will not include it in the commit snapshot. In the previous example, we used git add command to include the file for tracking.

$ git add README

In this example, we have now added the README file to include it for tracking. If we now run the git status command,

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

We can say that this file is now staged, as the message reads "Changes to be committed". If the file is committed now, the version of the file at the time of running the git add command will be there in the historical snapshot.

Staging Modified Files

Let’s say we already have a file named Testingfile.txt in our working repo which is a tracked file and we’ve done some changes to it. If we run the git status command it will give the following output.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: 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: Testingfile.txt

The recently modified file is under the section ‘Changes not staged for commit’. This means that the tracked file has been modified but not staged. To stage it, we run the git add command. git add is a multipurpose command, we can use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. After running the git add command, we’ll run the git status command again as follows:

$ git add Testingfile.txt

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: Testingfile.txt

Both the new file and modified file will go into the next commit. It is important to note that git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

Ignoring Files

We'll have a class of files that we don't want Git to add or show as untracked. These are generally automatically generated files such as log files or files produced by the build system. In such cases, we can create a file listing patterns to match them named: .gitignore. For example:

$ cat .gitignore

*.[oa]

*~

The first line tells Git to ignore any files ending in ".o" or ".a" — object and archive files that may be the product of building the code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. We can also include a log, tmp, or pid directory; automatically generated documentation; and so on.

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 (!).

Committing Changes

Once the staging area is set, we can commit the changes to the repository. The simplest way to commit the staged changes is by using git commit command.

$ git commit

The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will not change them unless it is explicitly asked to.

Commits can be thought of as snapshots or milestones along the timeline of a project. Commits are generally created with the git commit command to capture the state of a project at that point in time. Git Snapshots are always committed to the local repository. With Git, we’ll not interact with the central repository until we’re ready. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo.

The git commit option will open the text editor (customizable using git config), asking for a commit log message.

Please enter the commit message for your changes. Lines starting

with '#' will be ignored, and an empty message aborts the commit.

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: README

modified: Testingfile.txt

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what’s been changed. For example:

Change the message displayed by Testingfile.txt

- Changed the greeting message from Hello to Hi

- Added a ‘good bye’ message.

Comparing changes

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a Git command that runs a diff function on Git data sources. These data sources

can be commits, branches, files and more. The git diff command is used along with git status and git log to analyze the current state of a Git repo.

The git diff command compares files in the working directory with the ones in the staging area. The result highlights the changes that have been made, but not staged.

git diff --staged command is used to see what has been staged that will go into the next commit. The command compares the staged files with the last commit.

Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) 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. If you simply remove the file from your working directory, it shows up under the "Changes not staged for commit" (that is, unstaged) area of your git status output. Running git rm will then stage the removal of the file.

Moving files

Git doesn't explicitly track file movement. If you want to rename a file, you can run the following command:

$ git mv file_from file_to

Viewing the Commit History

To view the commit history of any project we'll use the git log command. Let's clone a project called Simple Calculator from GitHub.

$ git clone https://github.com/SimpleMobileTools/Simple-Calculator.git

After cloning the project, if we run the git log command, well get the following output:

```
$ git log
commit d54dafe8e193a4a6474e212559e5a48a11770e73 (HEAD -> master, origin/master, origin/HEAD)
Merge: 61e4f3c c7b9491
Author: Tibor Kaputa <tibor@kaputa.sk>
Date:   Tue Jan 15 22:43:31 2019 +0100

    Merge pull request #124 from ltGuillaume/patch-1

    Dutch
```



```
commit c7b949101500e2c77724df99ff3393f94287fd1f
Author: Guillaume <ltGuillaume@users.noreply.github.com>
Date:   Tue Jan 15 22:40:18 2019 +0100

    Dutch

commit 61e4f3cd0e3cfb28485d3f7638b8f045561aba37
Merge: 450265b 653c20d
Author: Tibor Kaputa <tibor@kaputa.sk>
Date:   Sun Jan 6 19:55:45 2019 +0100

    Merge pull request #121 from dugi991/master
```

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order. A lot of options are available with the git log command.

Undoing things

At any stage, if we want to undo something, we can do so in Git. This is one of the few areas in Git where we need to pay attention, as we may lose some work if we do it wrong. 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. This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message.

Unstaging a staged file

This can be achieved by using the git reset HEAD <file>....

$ git reset HEAD Testingfile.txt

Unstaged changes after reset:

M Testingfile.txt

$ 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: Testingfile.txt

At this point, the modified file is still unstaged.

Unmodifying a modified file

To discard the changes that we've made to a file, we use git checkout command.

Working with Remote Repositories

For collaborating any remote project, it is important to know how to manage remote repositories. Remote repositories are the ones that are hosted somewhere in a network. After cloning a repo, running git remote will tell you which remote servers you've configured. Example:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Enumerating objects: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0), pack-reused 1857
Receiving objects: 100% (1857/1857), 334.04 KiB | 155.00 KiB/s, done.
Resolving deltas: 100% (837/837), done.
```

$ cd ticgit

$ git remote

origin

You can also specify -v, which shows you the URLs that Git has stored for the short name to be used when reading and writing to that remote:

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

Adding Remote Repositories

To add a new remote Git repository as a short name you can reference easily, run:

git remote add <shortname> <url>

$ git remote

origin

$ 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)

Now we can use the string pb on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run git fetch pb:

$ git fetch pb
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Total 43 (delta 22), reused 22 (delta 22), pack-reused 21
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]   master   -> pb/master
 * [new branch]   ticgit   -> pb/ticgit

Fetching and Pulling from Remotes

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. by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to 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>.

If you want to push your master branch to your origin server, then you can run this to push any commits you've done back up to the server:

$ git push origin master

Inspecting a Remote

If you need more information about a remote, we can use git remote show <remote> command. If the same command is run with a particular short name, such as origin, you get something like this:

$ git remote show origin

* remote origin

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

Renaming and Removing Remotes

You can run git remote rename to change a remote's short name. For instance, if you want to rename pb to paul, you can do so with git remote rename:

$ git remote rename pb paul

$ git remote

origin

paul

If you want to remove a remote, you can either use git remote remove or git remote rm:

$ git remote remove paul

$ git remote

origin

Git Branching

Git doesn’t store data as series of changesets or differences, but as snapshots. A branch in Git is actually a simple file that contains the 40-character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

Branching and Merging

Look at the following workflow to understand branching and merging in Git.

1. Do some work on a website.
2. Create a branch for a new story you’re working on.
3. Do some work in that branch.

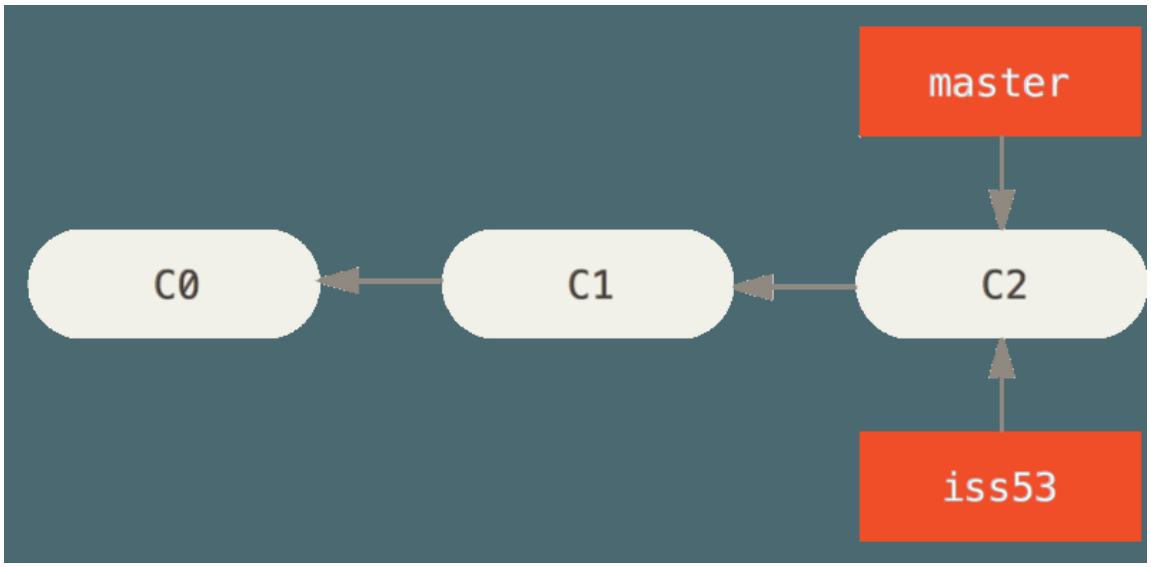
At this stage, you need to do another critical bug fix. You’ll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it’s tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

Assume that you project already has a couple of commits in the master branch. You need to work on some issue with an id (based on your issue tracking system). To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

$ git checkout -b iss53

Switched to a new branch "iss53"



You work on your website and do some commits. Doing so moves the iss53 branch forward, because you have it checked out:

$ git commit -a -m 'added a new footer [issue 53]'

Now you need to switch to a critical task as mentioned in the example above. With Git, you don’t need to deploy the fix along with the iss53 changes you’ve made, and you don’t have to revert those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your master branch.

$ git checkout master

Switched to branch 'master'

At this point, your project will be in the state as it was, before you started working on issue #53. When you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

We’ll now create a hotfix branch to work on.

$ git checkout -b hotfix

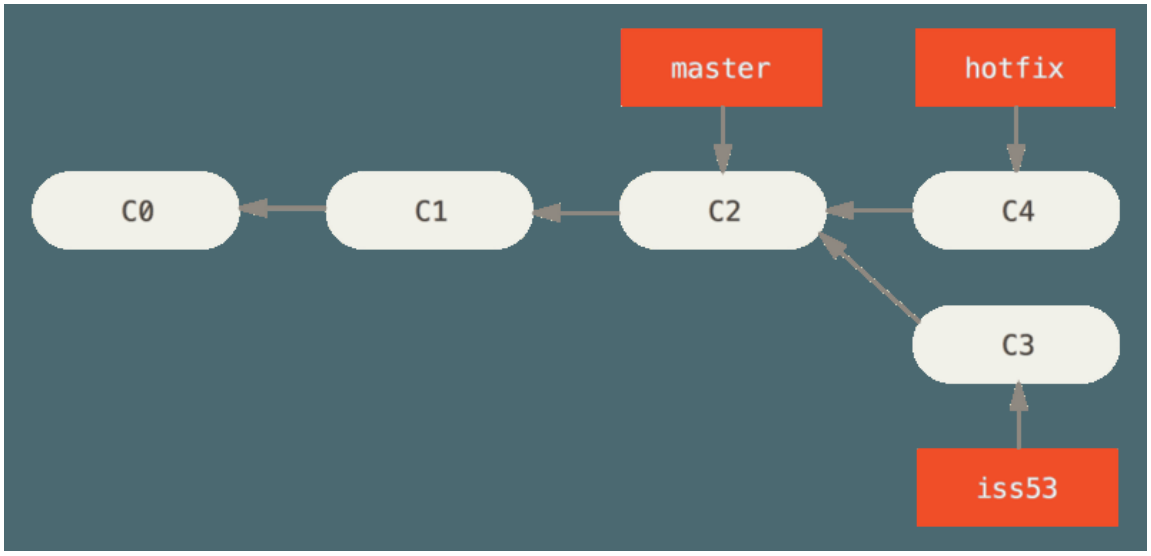
Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)



You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

$ git checkout master

$ git merge hotfix

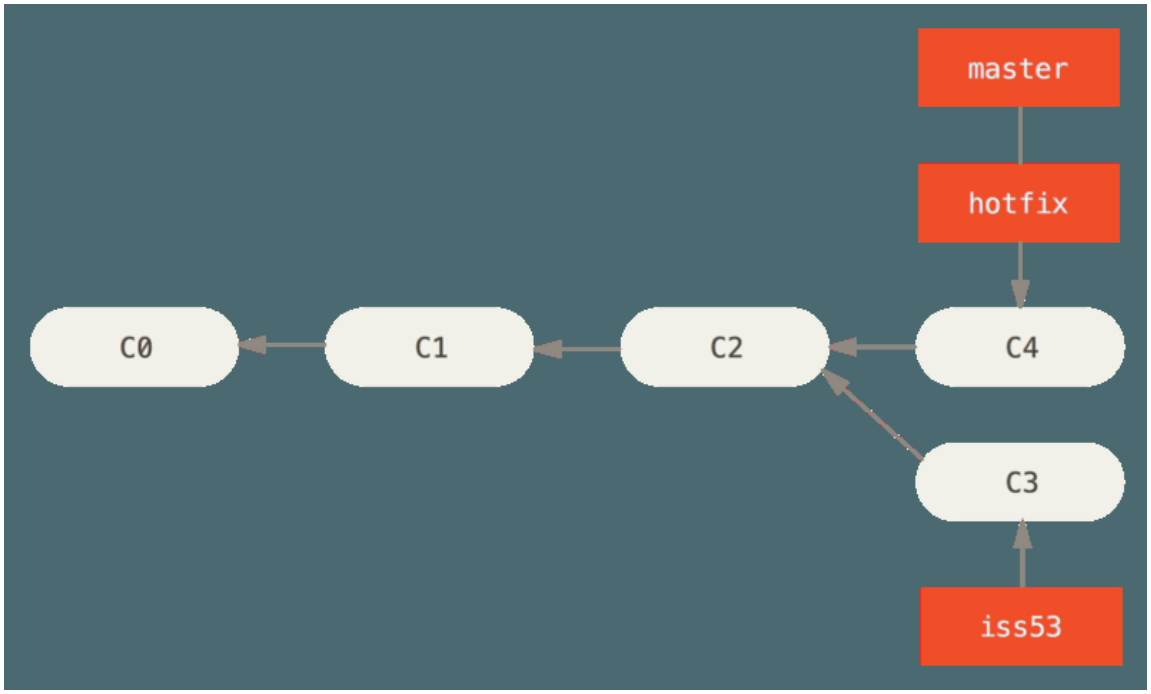
Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.



After the important fix is done, you can go back to what you were doing, ie, before you started working on the hotfix. You will first delete the hotfix branch and the master branch points at the same place.

$ git branch -d hotfix

Deleted branch hotfix (3a0874c)

You can then go back to your original issue (issue #53) and continue working on it.

$ git checkout iss53

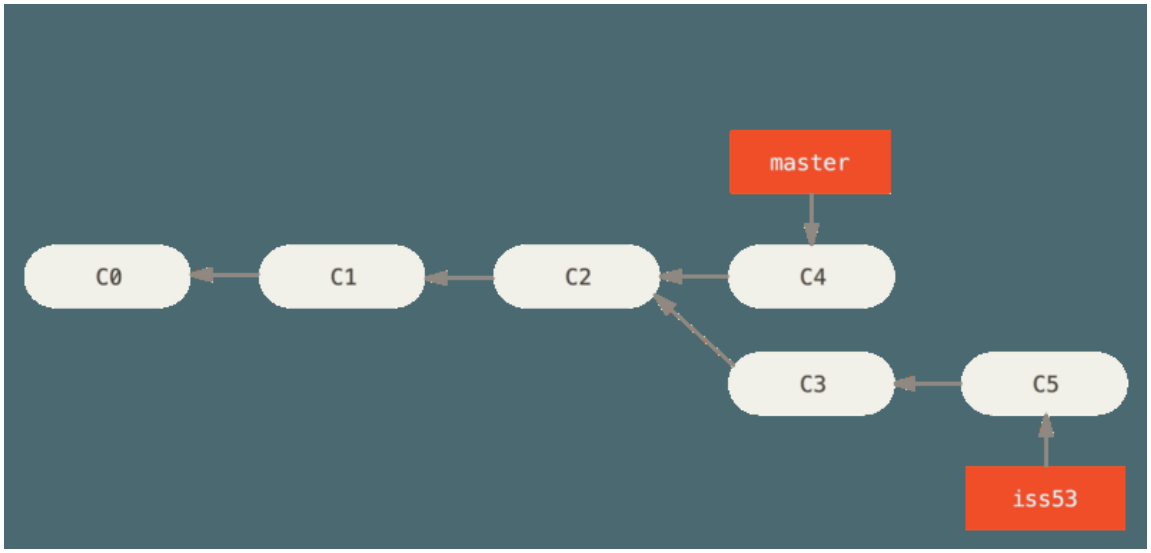
Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)



If you look at it, you'll see that the work done in the hotfix branch is not there in the files in iss53 branch. If you need to pull it in, you can merge your master branch into your iss53 branch by running git merge master, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

Merging

For example, if you want the work done in issue #53 to be merged into master, just checkout the branch and run the git merge command.

$ git checkout master

Switched to branch 'master'

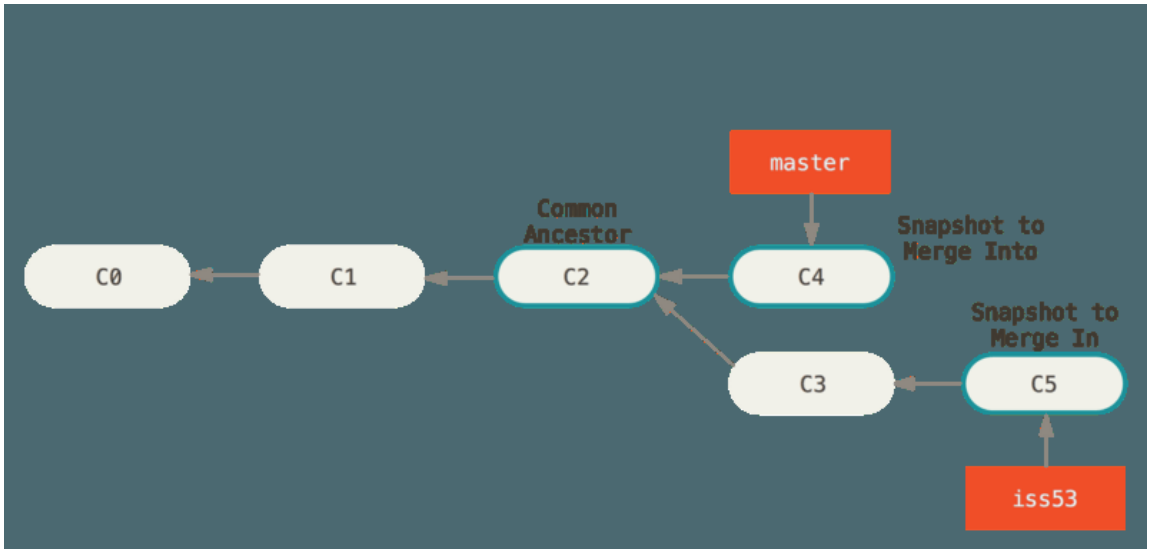
$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

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. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.



Merge Conflicts

In a practical scenario, merge conflicts may arise when two or more developers work on the same file in two different branches that are to be merged. Git will not be able to merge them cleanly. In the above example, if your fix for issue #53 modified the same part of a file as the hotfix branch, you'll get a merge conflict that looks something like this:

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git doesn't create a new merge commit. It will pause 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 status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

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

both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

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. Your file contains a section that looks something like this:

<<<<<<< HEAD:index.html

contact : email.support@github.com

=======

please contact us at support@github.com

>>>>>>> iss53:index.html

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =======), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

please contact us at email.support@github.com

Additional Reading: https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud