King Khalid University

Computer Science College

473-CSM Software Engineering II

Lab Manual

Version Control Systems and Git

‘

As an environmentalist I recommend that you don’t print this manual.
You really don’t need to, check it from your laptop whenever you need to, you’re a programmer so you’ll be using your laptop anyway.

Prepared by: Atheer Abdullah 2019

Lab Syllabus and Work plan.

Lab Number	Topics
1	What's version Control System. Types of VCSs.
2	What is Git? Files states and workflow,
3	Getting Started with Git, installing Git and configuring it.
4	Getting a Git Repository.
5	Recording Changes in a repo.
6	
7	Git branching
8	Tagging
9	Working with remotes. Github Pushing and Pulling
10	
11	
12	Documentation and writing a README file.
13	

Required Software

- Git: Instructions for installing the software is explained in Lab 3.
- An Editor, such as Notepad or VS Code.

References:

- ProGit-Everything you need to know about Git - by Scott Chacon and Ben Straub.
- Udacity Git courses: Version control using Git, optimize your Github, Writing READMEs

Assessment

	Marks	Time
6 Activities	2 Marks for each (1 extra activity)	After each lecture.
Project	10 Marks	Deadline: Monday 6th of April 2020

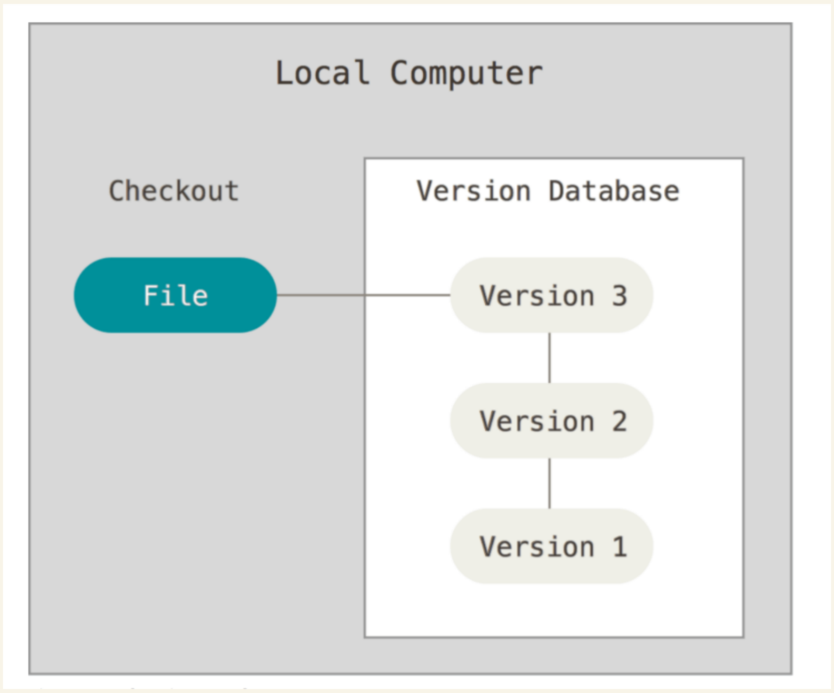
Table of Contents

What is Version Control System (VCS)?

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.
If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.

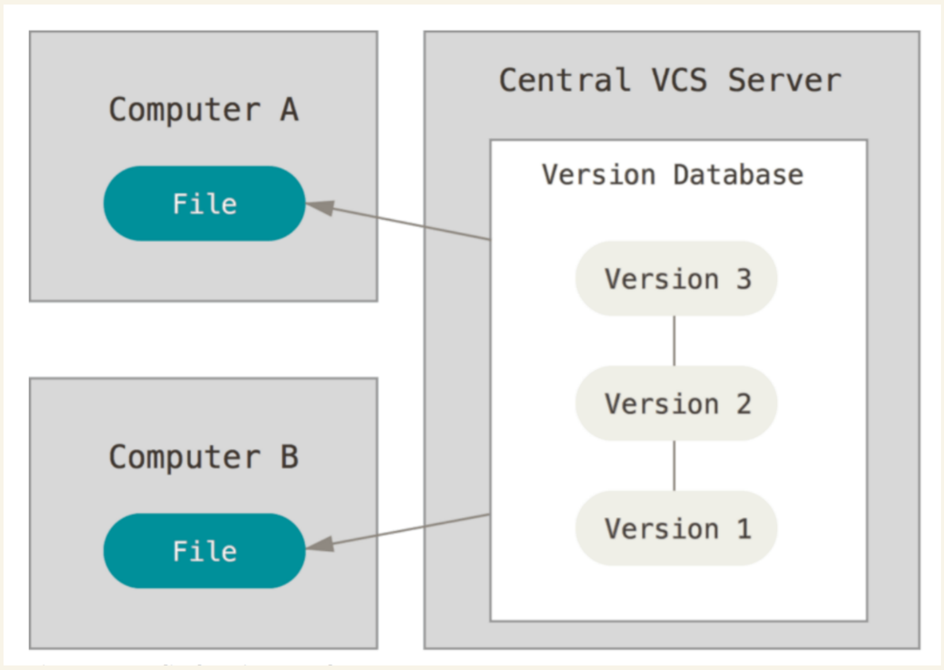
Local Version Control Systems

- Many people's version-control method of choice is to copy files into another directory.
- This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.
- To deal with this issue, programmers developed local VCSs that had a simple database that kept all the changes to files under revision control.



Centralized Version Control Systems

- The next major issue that people encounter is that they need to collaborate with developers on other systems.
- To deal with this problem, Centralized Version Control Systems (CVCSs) were developed.
- These systems have a single server that contains all the versioned files, and a number of clients that check out files from that central place.



- This setup offers many advantages:
    - For example, everyone knows to what everyone else on the project is doing.
- However, this setup also has some serious downsides. The most obvious is that it represents a single point of failure:
    - If that 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 of the central database got corrupted, and there are no proper backups, you lose everything.

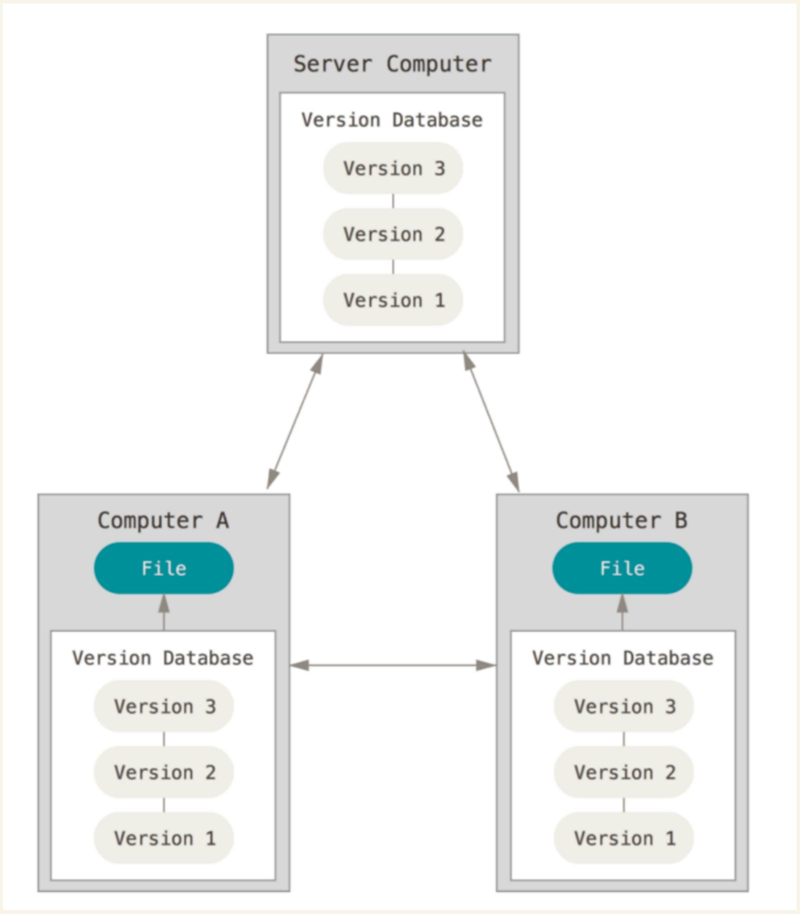
⚠ Local VCS systems suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in.

In a DVCS (such as Git), 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.
- ✓ In DVCS you can collaborate with different groups of people simultaneously within the same project.



What is Git?

- Git is a free and open source distributed version control system (DVCS).
- Git was invented by Linux creator Linus Torvalds to support the huge, disparate group of Linux developers.
- But Git's popularity is more closely tied to http://github.com. Git has been in existence for years but did not gain wide acceptance beyond the Linux community until the more recent surge in GitHub popularity.
- GitHub allows you to host open source projects at no cost. It also provides simple hooks and a friendly user experience that make Git easier to use.
- Git thinks of its data more like a series of snapshots of a mini filesystem.
- With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

Let's dive in

The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly.

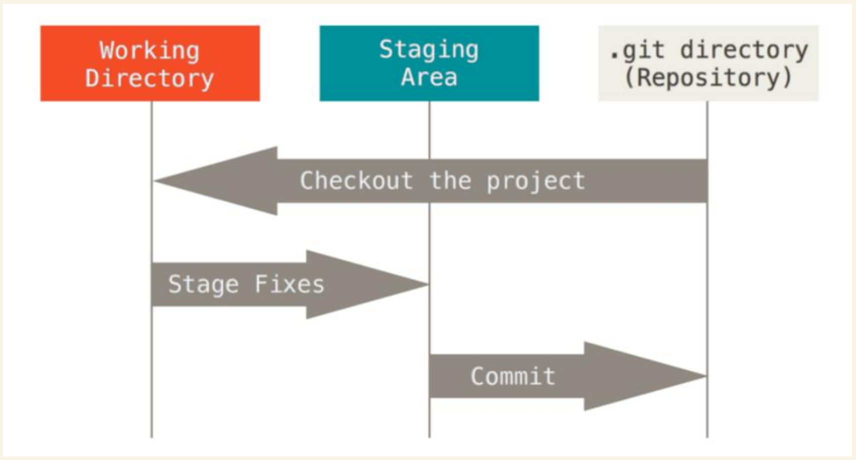
Git has three main states that your files can reside in: committed, modified, and staged:

✓ Committed means that the data is safely stored in your local database.

Modified means that you have changed the file but have not committed it to your database yet.

Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: **Git directory, working tree, staging area.**



1. The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

2. The working tree is a single checkout of one version of the project. These files are pulled out of the database in the Git directory and placed on disk for you to use or modify.

3. The staging area is a file, that stores information about what will go into your next commit.

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.

Let's Get Started

Installing Git

There are a lot of different ways to use Git. There are the original command-line tools, and there are many graphical user interfaces of varying capabilities.

For this course we'll be using the command-line, the command line is the only way to run all Git commands — most of the GUIs do not implement all of Git functionality to make it simple.

- Installing on Linux:
  - For different installing options, there are instructions for installing on several different Unix distributions on the Git website, at https://git-scm.com/download/linux.
- Installing on macOS
  - A macOS Git installer is maintained and available for download at the Git website, at https://git-scm.com/download/mac
- Installing on Windows
  - There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to https://git-scm.com/download/win and the download will start automatically.
  - Another easy way to get Git installed is by installing GitHub Desktop. The installer includes a command line version of Git as well as the GUI.

Customizing Git for the first time

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.

1. **Your Identity**

   The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information.

```
$ git config --global user.name "John Doe"

 $ git config --global user.email johndoe@example.com
```

--global option means that you need to do this only once, because then Git will always use that information for anything you do on that system.

2. Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message**. If not configured, Git uses your system's default editor.**

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file.

Example:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"

$git config --global core.editor "'C:\Program Files\Microsoft VS Code\code.exe' -n -w"
```

3. Checking Your Settings

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

```
$ git config  --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
 ...
```

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

```
$ git config user.name
 John Doe
```

Getting Help

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

```
$ git help <verb>
```

OR

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

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

```
$ git help config
```

If you don't need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can use the -h or --help options, as in:

```
$ 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
    --renormalize         renormalize EOL of tracked files (implies -u)
    -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
```

Getting a Git Repository

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

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

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

Initializing a Repository in an Existing Directory

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

for Linux:

```
$ cd /home/user/my_project
```

for macOS:

```
$ cd /Users/user/my_project
```

for Windows:

```
$ cd /c/user/my_project
```

and type:

```
$ git init
```

<u>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.</u>

Cloning an Existing Repository

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

For example, if you want to clone the Git linkable library called "libgit2", you can do so like this:

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

This command will:

1. Create a directory named libgit2,
2. Initialize a .git directory inside it,
3. Pull down all the data for that repository, and check out a working copy of the latest version. If you go into the new libgit2 directory that was just created, you'll see the project files in there, ready to be worked on or used.

Recording 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.

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

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

Tracking New Files

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

```
$ git add README
```

If you want to add all the files in the directory you're currently in you can use:

```
$ git add .
```

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ 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
```

Staging Modified Files

Let's change a file that was already tracked.

If you change a previously tracked file and then run your git status command again you will find the file under "Changes not staged for commit "section.

If you want to stage this file you can run

```
$ git add <file name>
```

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.

Viewing Your Staged and Unstaged Changes

- If you want to know exactly what you changed, not just which files were changed — you can use the git diff command.
- If you want 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.

💡 If you've staged all of your changes, git diff will give you no output. But you can use it to see what you've changed but not yet staged.

Committing Your Changes

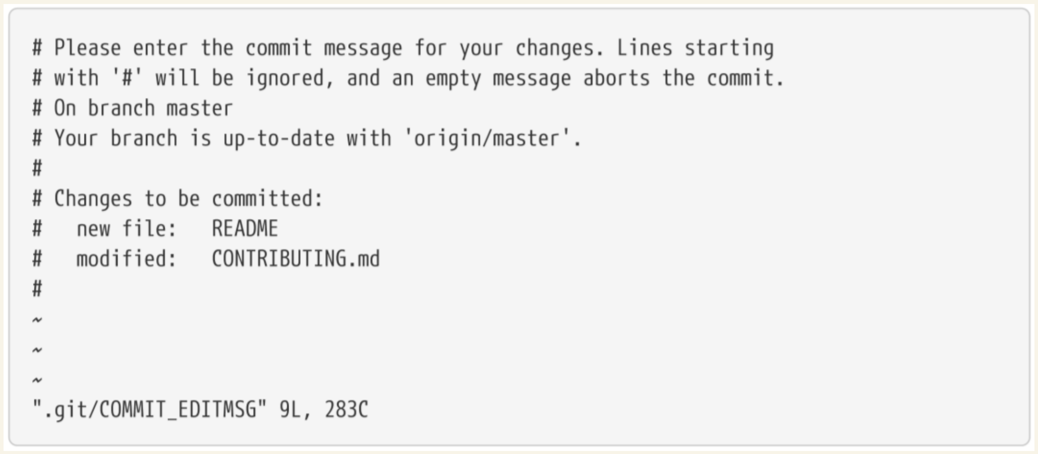
Now that your staging area is set up the way you want it, you can commit your changes.

💡 **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.**

The simplest way to commit is to type git commit:

```
$ git commit
```

Doing so launches your editor of choice. This is an example using vim editor.



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing.

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

```
$ git commit -m "Change header color "
```

Now you've created your first commit!

You can see that the commit has given you some output about itself: which branch you committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.

What's SHA-1 checksum?

SHA-1 hash is the mechanism that Git uses for this checksumming. It is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

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.

Removing Files

To remove a file from Git, you can use git rm command.

```
$ git rm <file name>
```

If you modified the file or had already added it to the staging area, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Renaming files

If you want to rename a file you can use the command mv

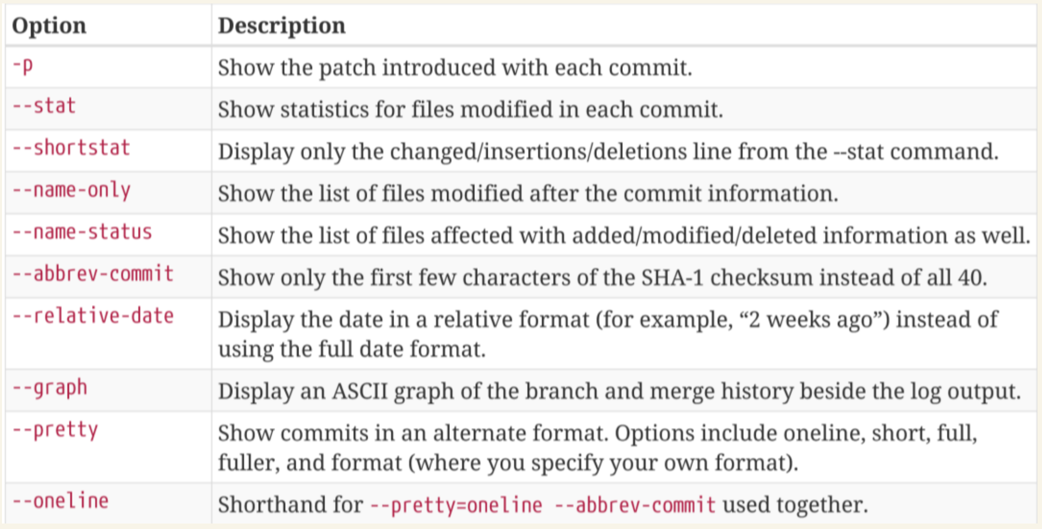
```
$ git mv file_from file_to
```

Viewing the Commit History

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

It lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first.

If you're looking for specific output, you can do so by using different flags, here are some common useful ones.



Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the --stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

You can also view the log history of a specific time, using time-limiting options such as --since and --until are very useful. For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

Undoing Things

At any stage, you may want to undo something.

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.

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,

1. Make the additional changes you forgot,
2. Stage them,
3. Commit again using the --amend option.

Branching

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

A branch in Git is simply a movable pointer to one of your commits. The default branch name in Git is master.

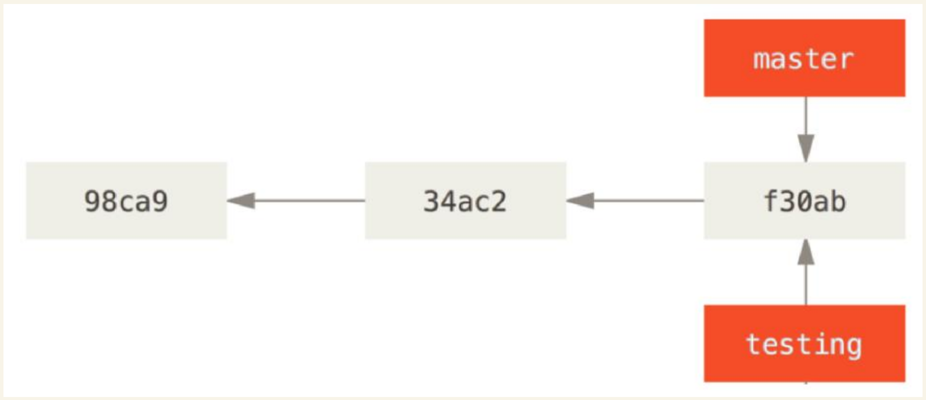
As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you want to create a new branch called testing. You do this with the git branch command:

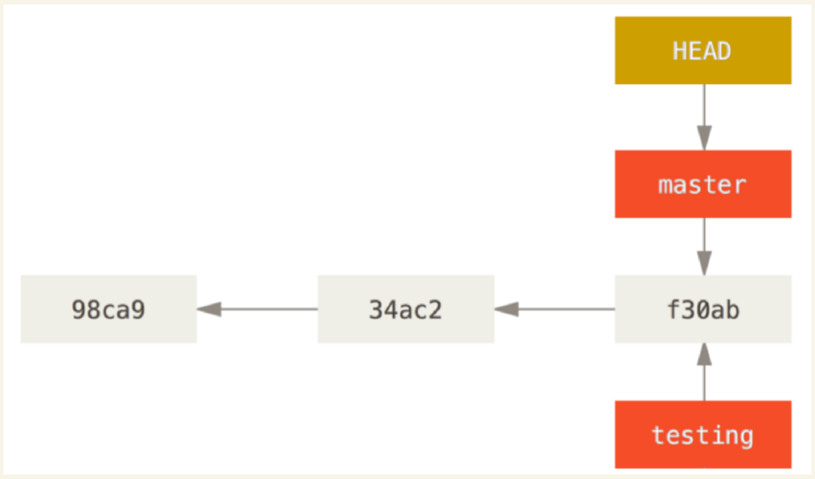
```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on. Which is– by default- "master".



? How does Git know what branch you're currently on? It keeps a special pointer called HEAD.

In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master. **The git branch command only created a new branch — it didn't switch to that branch.**



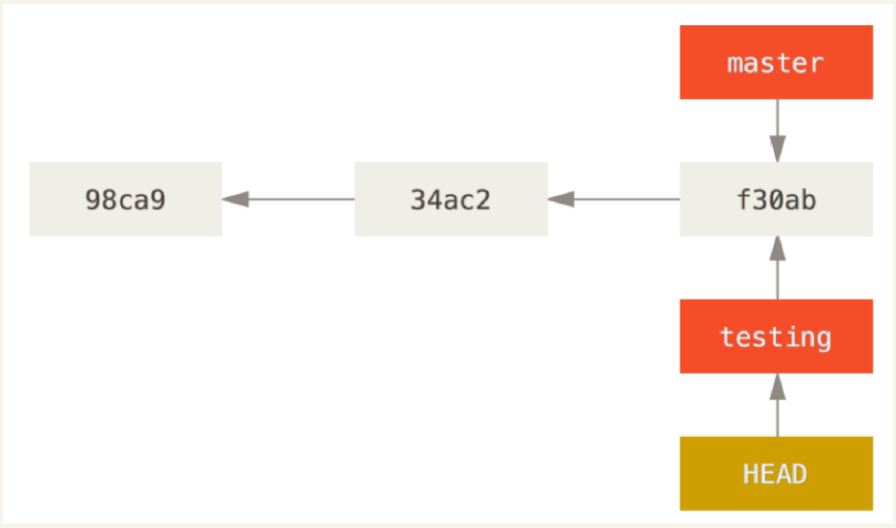
You can use git log command to show you where the branch pointers are pointing.

Switching Branches

To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

```
$ git checkout testing
```

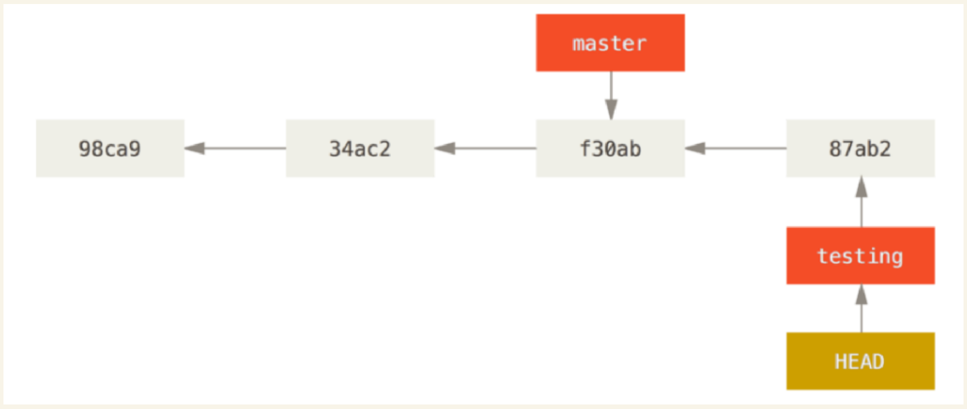
This moves HEAD to point to the testing branch.



? **What is the significance of that?**

Well, let's say we made another commit:

```
$ git commit -a -m 'made a change'
```

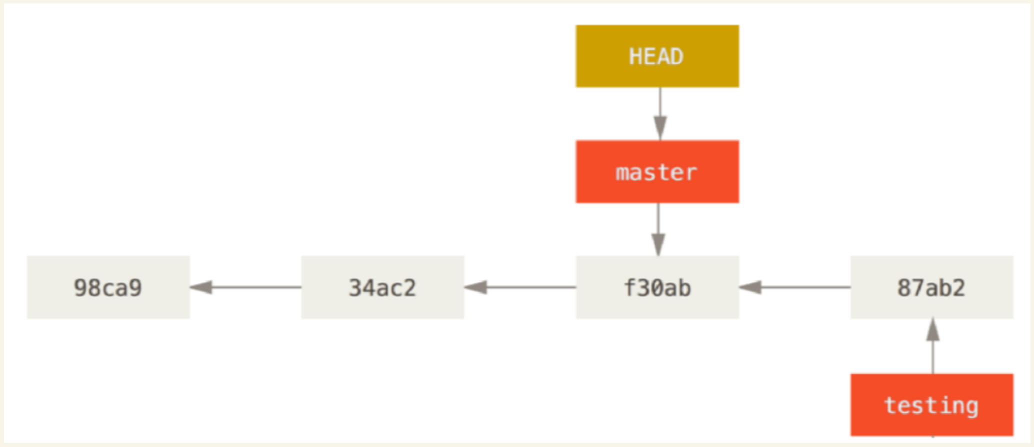


This is interesting, because now your testing branch has moved **forward**, but your master branch still points to the commit you were on when you ran git checkout to switch branches.

Let's switch back to the master branch:

```
$ git checkout master
```

What do you think will happen?



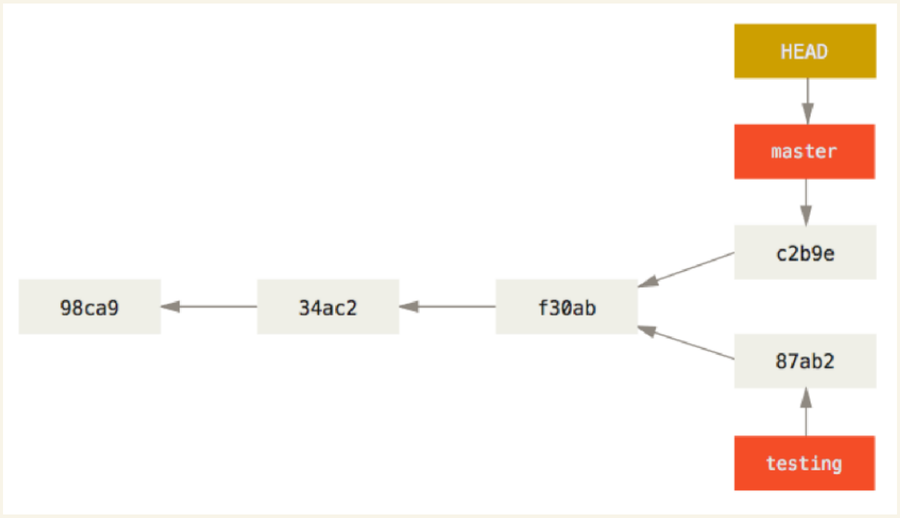
✓ Correct , The head points now to "master" branch.

That command did two things:

1. It moved the HEAD pointer back to point to the master branch,
2. It reverted the files in your working directory back to the snapshot that master points to. <u>This also means the changes you make from this point forward will diverge from an older version of the project.</u>

Now, when you make a commit using the "master" branch, your new commits will start from the last commits in the "master" branch. Which has nothing to do with the "testing" latest commit.

And as you can see, we have two lines or branches of changes or commits.



How does this help?

- Let's say you were working on your website and did some commits on your master branch.
- You published your website,
- Then you thought of adding another feature to your website, you made a new branch for this feature.
- As you were working on it, you got an issue with the website you have already published, and you need to fix it.
- Yet you're not done with the new feature and you can't publish it right now, so what should you do?
- Should abandon all of your work? No!
- You could just switch back to the master branch, and fix the issue,
- You can come back to the new branch whenever you want to work on the new feature.

Merging branches

Now, let's say you fixed the issue, and you're done with the new feature, and you want all of this work to be combined together, you can merge the master branch with the new branch.

First you need to switch to the master branch by:

```
$git checkout master
```

And then use git merge <branchname> command, assume that your new branch is called "testing"

```
$git merge testing
```

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).

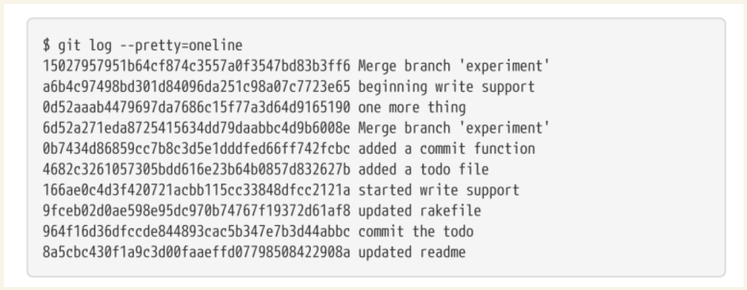
Git supports two types of tags:

- **Lightweight Tags:** A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit.
  To create a lightweight tag use the command tag followed by the name of the tag you want

  ```
  $git tag v1.0
  ```
- **Annotated tags:** are stored as full objects in the Git database. They contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

Tagging Later

You can also tag commits after you've moved past them. Suppose your commit history looks like this:



```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Now, suppose you forgot to tag the project at v1.2, which was at the "updated rakefile" commit. You can add it after the fact. 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
```

Listing your tags:

```
$ git tag
```

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.
- You can have several of them, each of which generally is either read-only or read/write for you.
- Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

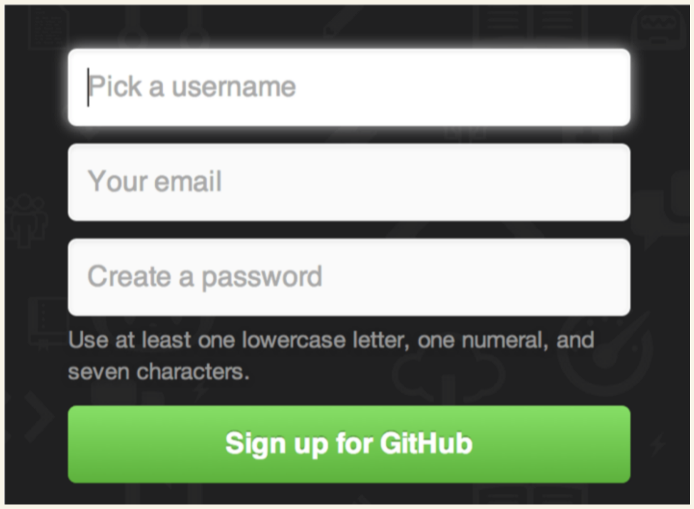
GitHub

GitHub is the 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.

Account Setup and Configuration

The first thing you need to do is set up a free user account. Simply visit https://github.com, choose a username that isn't already taken, provide an email address and a password, and click the big green "Sign up for GitHub" button.

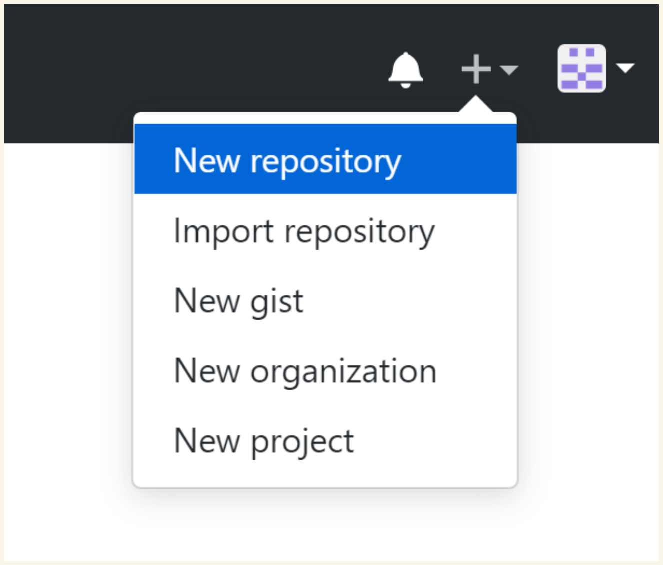

Pick a username
Your email
Create a password
Use at least one lowercase letter, one numeral, and seven characters.
Sign up for GitHub

Check out this course to strengthen your presence on Github.

Now you're ready to create your own new repositories and to clone other's repos. You can also contribute to other projects.

Creating a New Repository

Let's create a new repository to share our project code with. Start by clicking the "New repository" button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your username as seen in The "New repository" dropdown.



All you have to do now is provide a **project name**; the rest of the fields are completely optional.

For now, just click the "Create Repository" button, and boom – you have a new repository on GitHub, named <user>/<project_name>.

Since you have no code there yet, GitHub will show you instructions for how to create a brand-new Git repository or connect an existing Git project.

Adding Remote Repositories

Now, you made a new repository on your Github account, and you want to push your local repository to it, you first have to add this remote server that you're pushing to. To add a new git remote repository, you use the command git remote add <shortname> <url>

```
$git remote add origin https://github.com/*********/test.git
```

- shortname, is any name you want, the default that git uses is "origin".
- url: is the url to the new repository you just made, and you want to push to.

Renaming and Removing Remotes

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

For instance, if you want to rename origin to new, you can do so with git remote rename:

```
$ git remote rename origin new
```

If you want to remove a remote for some reason, you can either use git remote remove

```
$ git remote remove new
```

Pushing and Pulling

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it to the server. The command for this is simple: git push <remote> <branch>.

💡 Default branch name is "master".

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

```
$ git push origin master
```

Again:

- "Origin" is the shortname for the remote repository.
- "Master" is the branch you're currently working on, on your local machine.

What happens when you use this command, is that all of your commits on this branch will be uploaded to the server.

So, each time you make enough commits on your work, and you want to share it, you can just use this command and it will be available on your remote repository.

Fetching and Pulling from Your Remotes

To get data from your remote projects, you can run:

```
$ 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.

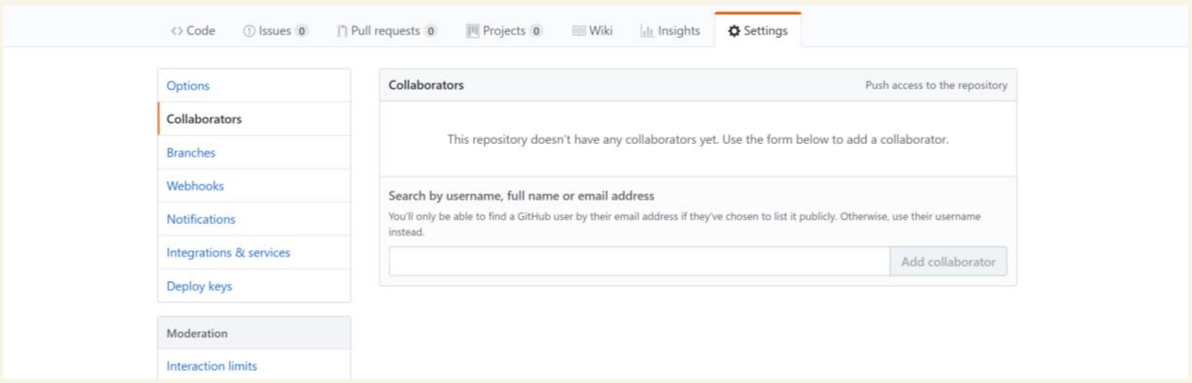
It's important to note that the **git fetch** command only downloads the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

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

Adding Collaborators

If you're working with other people who you want to give commit access to, you need to add them as "collaborators".

Click on Settings >> Collaborators >> type in the collaborator email or username.



This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

Search by username, full name or email address

You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.

Contributing to a Project

If you want to contribute to an existing project to which you don't have push access, you can "fork" the project. When you "fork" a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, and you can push to it.

If you believe that your commits are helpful, you can make a pull request.

You can discuss those changes with the community, If the repository owner wants to incorporate your changes, they will merge your commits with theirs.

Documentation and README files

When you hear the phrase technical documentation you might think of dry literature that is difficult to understand. But good documentation isn't boring or unapproachable -- often it's written in plain English!

Sometimes, it's even written as a series of guides or tutorials.

As it turns out, there isn't a secret ruling body that makes laws for what documentation should and shouldn't look like**. That's because unlike your code which is written for computers documentation is written for humans.** Documentation exists to help us make sense of the code that we've written, which may not always be quite as intuitive as we'd like it to be.

For instance, if you're working a large open source library that other people can use, good documentation is absolutely essential for acquiring both users and contributors. If your documentation is good, people will want to use your library in their projects. If your documentation is great, people might even chip in and help you with your library.

What's a README file?

A README is a text file that introduces and explains a project. It contains information that is commonly required to understand what the project is about.

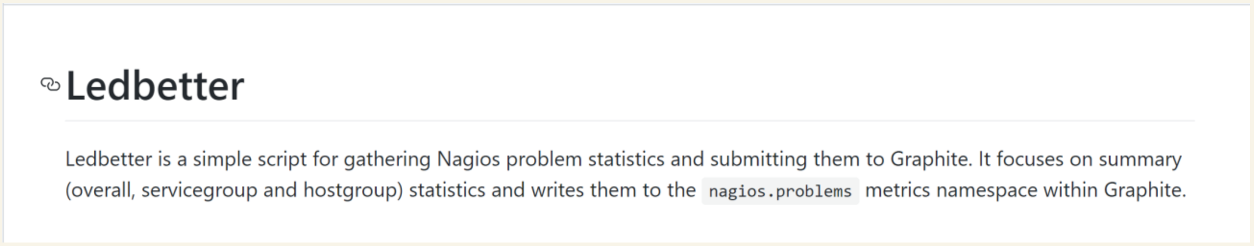
Writing your own README file

Let's take a look at this well-documented example to learn how to write a good README file,

https://github.com/github/ledbetter

A readme should provide just enough context to get a user up and running with your code. Keep in mind, we are writing this for other humans.

1. **Description:** We'll start with a title and a description.

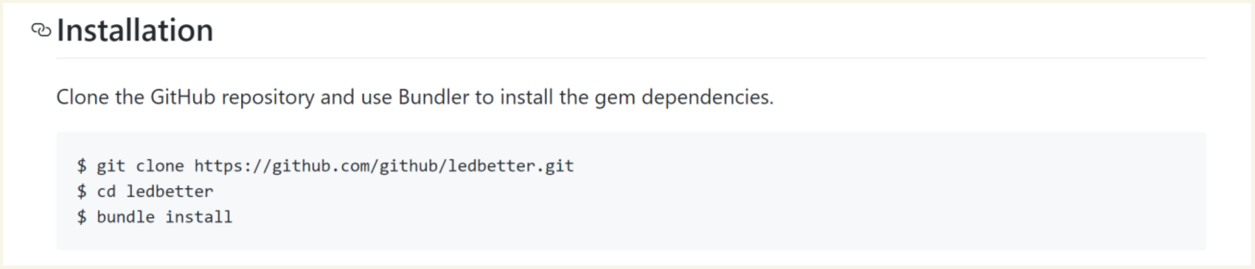


Ledbetter

Ledbetter is a simple script for gathering Nagios problem statistics and submitting them to Graphite. It focuses on summary (overall, servicegroup and hostgroup) statistics and writes them to the `nagios.problems` metrics namespace within Graphite.

A sentence or two will work just fine but be sure to capture the spirit of your project clearly and concisely. This will help frame the reader's experience when going through your documentation.

2. **Installation:** Next, include any information that is absolutely necessary for understanding your code. This may be dependencies on other software or libraries, installation instructions, common usage, or known bugs.



Installation

Clone the GitHub repository and use Bundler to install the gem dependencies.

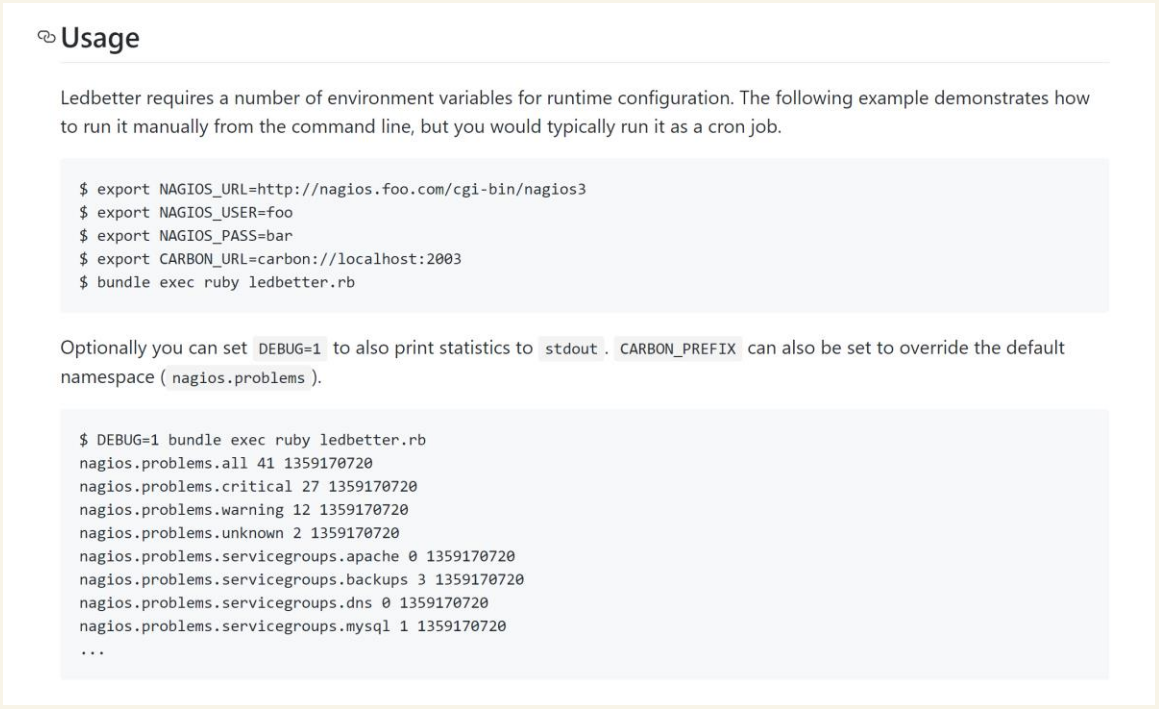
```
$ git clone https://github.com/github/ledbetter.git
$ cd ledbetter
$ bundle install
```

The important part is that you communicate clearly and concisely any information that is essential for the user. And that you don't make any assumptions about what the end user already knows.

It's up to you to decide what is and isn't essential information. A few good questions to ask yourself. Include, what are the exact steps that need to be taken to get this code base up and running?

What should they already have installed or configured? What might they have a hard time understanding right away?

3. **Usage:** Often, developers will include a getting started, or an installation section to help with any initial setup that may need to happen before using your code. If it's helpful, include example code to better illustrate your end user's proper usage of your code.



Usage

Ledbetter requires a number of environment variables for runtime configuration. The following example demonstrates how to run it manually from the command line, but you would typically run it as a cron job.

```
$ export NAGIOS_URL=http://nagios.foo.com/cgi-bin/nagios3
$ export NAGIOS_USER=foo
$ export NAGIOS_PASS=bar
$ export CARBON_URL=carbon://localhost:2003
$ bundle exec ruby ledbetter.rb
```

Optionally you can set `DEBUG=1` to also print statistics to `stdout`. `CARBON_PREFIX` can also be set to override the default namespace (`nagios.problems`).

```
$ DEBUG=1 bundle exec ruby ledbetter.rb
nagios.problems.all 41 1359170720
nagios.problems.critical 27 1359170720
nagios.problems.warning 12 1359170720
nagios.problems.unknown 2 1359170720
nagios.problems.servicegroups.apache 0 1359170720
nagios.problems.servicegroups.backups 3 1359170720
nagios.problems.servicegroups.dns 0 1359170720
nagios.problems.servicegroups.mysql 1 1359170720
...
```

It's important to be mindful of the assumptions you make about what the user's expected to know.

Remember, you're familiar with your code already. So, think about it from the perspective of someone who's never seen this before.

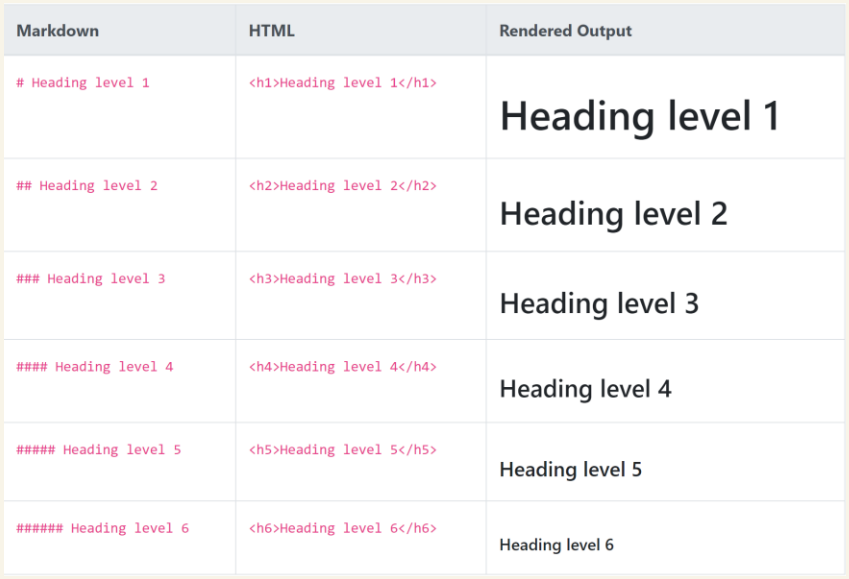
4. **Support:** Tell people where they can go to for help. It can be any combination of an issue tracker, a chat room, an email address, etc.
5. **Contributing:** State if you are open to contributions and what your requirements are for accepting them.
6. **Authors and acknowledgment**: Show your appreciation to those who have contributed to the project.
7. **License**: For open source projects, say how it is licensed.
8. **Project status**: If you have run out of energy or time for your project, put a note at the top of the README saying that development has slowed down or stopped completely. Someone may choose to fork your project or volunteer to step in as a maintainer or owner, allowing your project to keep going. You can also make an explicit request for maintainers.

Markdown and README file

While READMEs can be written in any text file format, the most common one that is used nowadays is Markdown. It's a markup language that allows you to add some lightweight formatting.

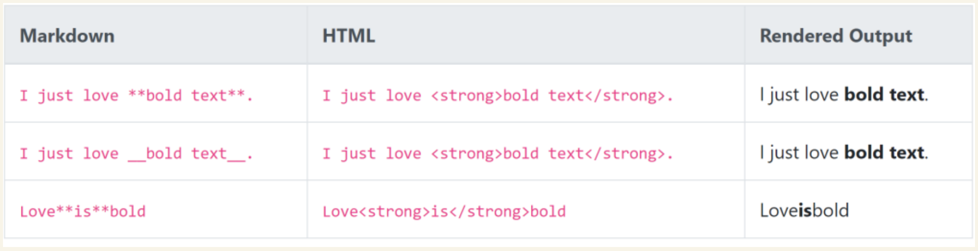
Using Markdown doesn't mean that you can't also use HTML. You can add HTML tags to any Markdown file.

Headings: To create a heading, add number signs (#) in front of a word or phrase. The number of number signs you use should correspond to the heading level. For example, to create a heading level three (<h3>), use three number signs (e.g., ### My Header).



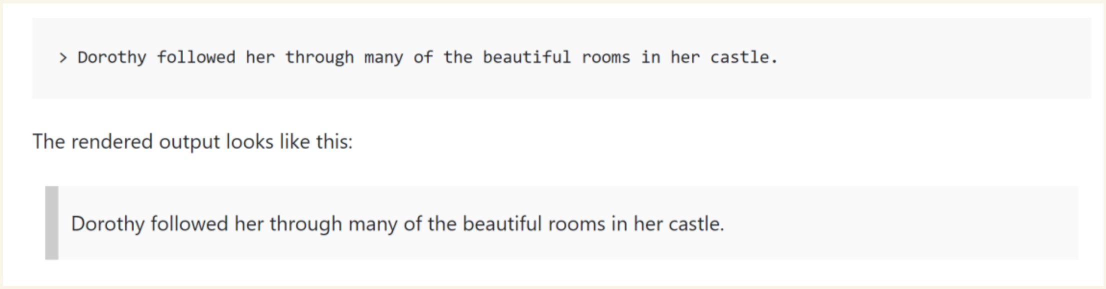
Markdown	HTML	Rendered Output
`# Heading level 1`	`<h1>Heading level 1</h1>`	Heading level 1
`## Heading level 2`	`<h2>Heading level 2</h2>`	Heading level 2
`### Heading level 3`	`<h3>Heading level 3</h3>`	Heading level 3
`#### Heading level 4`	`<h4>Heading level 4</h4>`	Heading level 4
`##### Heading level 5`	`<h5>Heading level 5</h5>`	Heading level 5
`###### Heading level 6`	`<h6>Heading level 6</h6>`	Heading level 6

Bold: To bold text, add two asterisks or underscores before and after a word or phrase. To bold the middle of a word for emphasis, add two asterisks without spaces around the letters.



Markdown	HTML	Rendered Output
`I just love **bold text**.`	`I just love <strong>bold text</strong>.`	I just love **bold text**.
`I just love __bold text__.`	`I just love <strong>bold text</strong>.`	I just love **bold text**.
`Love**is**bold`	`Love<strong>is</strong>bold`	Love**is**bold

Blockquotes: To create a blockquote, add a > in front of a paragraph.



```
> Dorothy followed her through many of the beautiful rooms in her castle.
```

The rendered output looks like this:

> Dorothy followed her through many of the beautiful rooms in her castle.

Learn more about Markdown markup language here.

Learn more about writing a README file here.

- ✓ Now you're a GitHub user.
- ✓ You know how to create and manage your account,
- ✓ Create and push to repositories,
- ✓ Contribute to other people's projects and accept contributions from others.

VCS management is a very important skill that would help you a lot as a developer, and You're on your way to make wonderful projects, Keep going!✨

Project Guidelines:

Prerequires:

- Deploy any of the following:
    - Your own ISM362 project.
    - Any other project you worked on, a website, a mobile application … etc.
    - Clone and download any project you want to contribute to on Github.
    - Or sign-in using your Github account to https://codepen.io/ and choose any project you like, download it and start a new local reop inside the downloaded folder.
- Having your own Github account.

Requirements:

- Initialize a new repository in the project folder/forking a repo [1 Mark]
- Commit all the files in your project and name it as "initial commit" [1 Mark]
- Create a new repository in your Github account. [1 Mark]
- Push your local repo to your remote one. [2 Marks]
- Make 4 other changes to your project and commit each one separately and give each commit a comment. [2 Marks]
- Push the local repo to your remote one with each commit.

- Make a Formatted (using Markdown) README file with: [2 Marks]
    - A clear description of your project.
    - Installation guide of every required tool or software for your project.
    - Some examples of using your code.
    - OPTIONAL: Extra details about getting support or contributing to your project.
- Edit your Github profile, write a bio. [1 Mark]
    - Check this course out.

How to submit?

Make a .doc file that contains:

- Screenshots of each of the commands you used in each step.
- The link to your project on Github.

All the best.
Atheer Abdullah