

A photograph of a person's hands typing on a laptop keyboard. The laptop screen shows a grid of icons. In the background, a computer monitor displays a dark interface, and there are some books or papers on a shelf.

Comprehensive GIT using GitLab



TABLE OF CONTENTS

1. Git and GitLab Introduction and Basics: 3
2. GitLab Flow: 21
3. Branching: 88
4. Configuring Git: 105
5. Rebasing: 110
6. Merging: 130
7. Resolving Merge Conflicts: 155
8. Remote Repositories: 162
9. Reviewing the Commit History: 185
10. Improving Your Daily Workflow: 192
11. Continuous Integration / Continuous Delivery (CI/CD): 226

1. Git and GitLab Introduction and Basics



Introducing GitLab

In this lesson, we'll explore the following topics:

- An overview of version control
- The main features of GitLab
- Self-managed versus SaaS
- Free versus paid
- A brief history of GitLab

Version control systems and Git

- Let's say you write code, or work on a course, or even just want to collect and update a set of text-based documents.
- You need some method of keeping track of changes, of being able to revert mistakes in the work, or branch in new directions; and you'll probably want some way of remotely backing up your work in case of fire, theft, or acts of a misbehaving computer.

GitLab and Git

- GitLab is built on top of git so that users who are contributing work (editing code, writing lessons, and so on) to a project will have a copy of the project downloaded/checked out/cloned on their local computer.
- It provides a web interface for handling many of git's more advanced workflows, and recommends a workflow for interacting with git for the best in productivity, efficiency, and ease of use.

Features

- There is the classic file browser that lets you explore the files in your repository:

📁 shared	Add GitLab Pages	1 year ago
📁 spec	Merge remote-tracking branch 'dev/master'	10 hours ago
📁 symbol	Resolve "Better SVG Usage in the Frontend"	10 months ago
📁 tmp	Move Prometheus presentation logic to Prometh...	1 year ago
📁 vendor	Specify Jupyter Image to use with JupyterHub In...	6 days ago
📄 .babelrc	only apply rewire plugin when running karma tests	3 months ago
📄 .codeclimate.yml	Removed API endpoint and specs	1 month ago
📄 .csscomb.json	Remove SCSS rules for short hex chars.	1 year ago
📄 .eslintignore	update eslintignore for node scripts	4 months ago
📄 .eslintrc.yml	Enable "prefer-destructuring" in JS files	1 month ago
📄 .flayignore	Backport from EE !5954	3 weeks ago
📄 .foreman	complete hooks for post receive	6 years ago
📄 .gitattributes	Start to use Danger for automating MR reviews	2 weeks ago
📄 .gitignore	Exclude Geo DB Yaml on CE too	1 month ago

Features

- There's also a branch viewer, which lets you see variations of your work under active development, as well as branches that are considered stale and no longer developed:

The screenshot shows the 'Branches' page in the GitLab Community Edition. At the top, there are navigation links: GitLab.org > GitLab Community Edition > Repository > Branches. Below this is a navigation bar with tabs: Overview (which is selected), Active, Stale, and All. To the right of the tabs is a search bar labeled 'Filter by branch name'. The main content area is titled 'Active branches' and lists five branches:

- 48773-gitlab-project-import-should-use-object-storage**: This branch has a commit history showing a merge from 'rails5-update-gemfile-lock-2' into 'master' at commit `a8c87544`, made 5 minutes ago. It has 56 pushes and 8 merges. Action buttons include 'Compare' and a dropdown menu.
- bugs-and-regressions-process**: This branch has a commit history showing a merge from 'rails5-update-gemfile-lock-2' into 'master' at commit `46a54123`, made 13 minutes ago. It has 2 pushes and 10 merges. Action buttons include 'Compare' and a dropdown menu.
- artifact-format-v2-with-parser**: This branch has a commit history showing a merge from 'rails5-update-gemfile-lock-2' into 'master' at commit `fda5e9bd`, made 40 minutes ago. It has 2 pushes and 203 merges. Action buttons include 'Compare' and a dropdown menu.
- master**: This branch is marked as default and protected. It has a commit history showing a merge from 'rails5-update-gemfile-lock-2' into 'master' at commit `835bacc2`, made 51 minutes ago. Action buttons include a dropdown menu.
- 48419-charts-with-long-label-appear-oversized**: This branch has a commit history showing a merge from 'rails5-update-gemfile-lock-2' into 'master' at commit `835bacc2`, made 51 minutes ago. It has 0 pushes and 0 merges. Action buttons include 'Compare' and a dropdown menu.

At the bottom of the list, there is a link 'Show more active branches'.

Features

- Alongside this is a tag viewer that lets you explore specific releases of your work:

Tags give the ability to mark specific points in history as being important

 **v10.8.7** protected Version v10.8.7

-o [eb600b0b](#) · Update VERSION to 10.8.7 · 1 day ago

 **v11.0.5** protected Version v11.0.5

-o [a4583c3b](#) · Update VERSION to 11.0.5 · 1 day ago

 **v11.1.2** protected Version v11.1.2

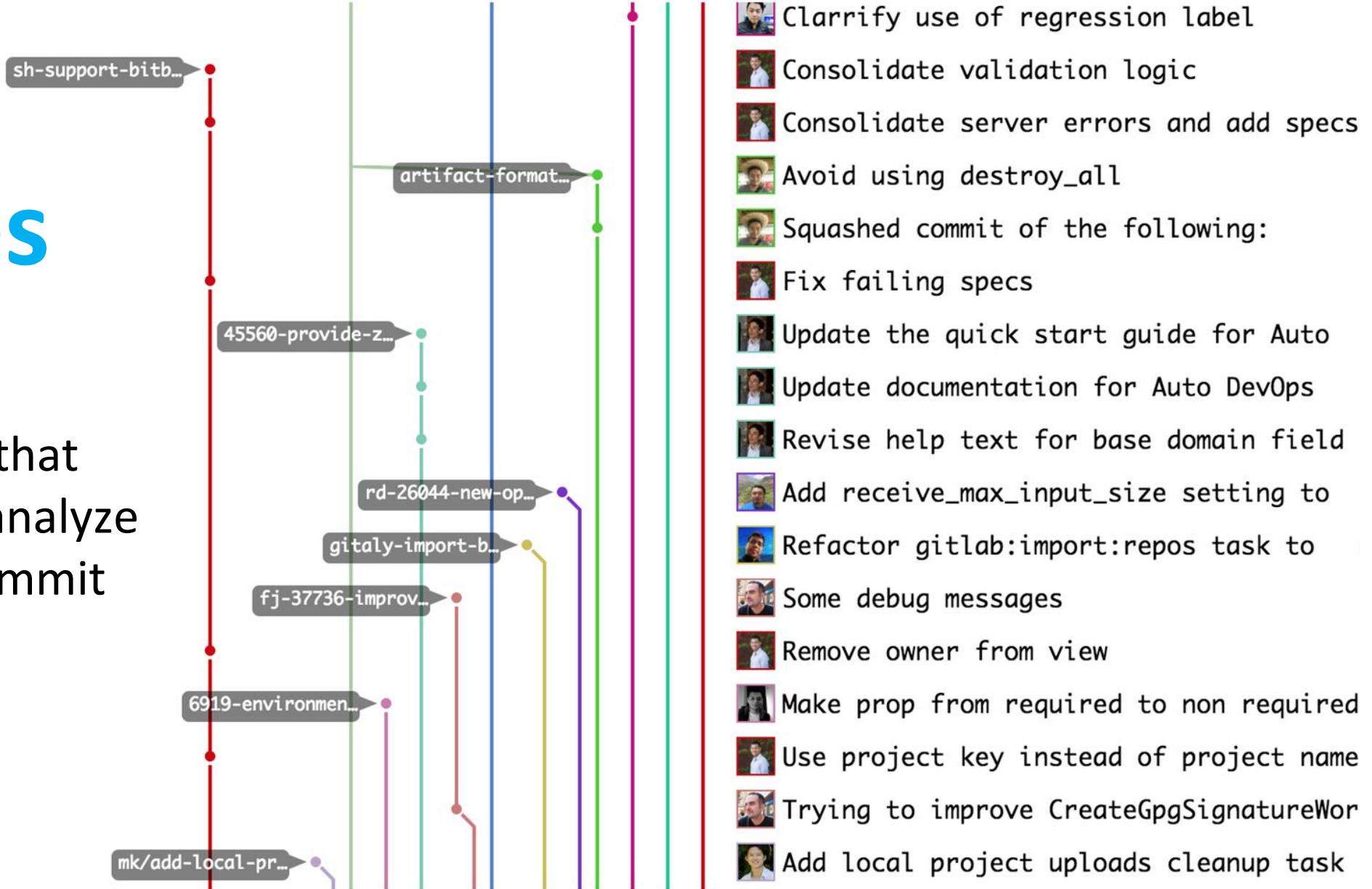
-o [35936b0b](#) · Update VERSION to 11.1.2 · 1 day ago

 **v11.1.1** protected Version v11.1.1

-o [94b93230](#) · Update VERSION to 11.1.1 · 3 days ago

Features

- There are tools that can be used to analyze and view the commit graph:

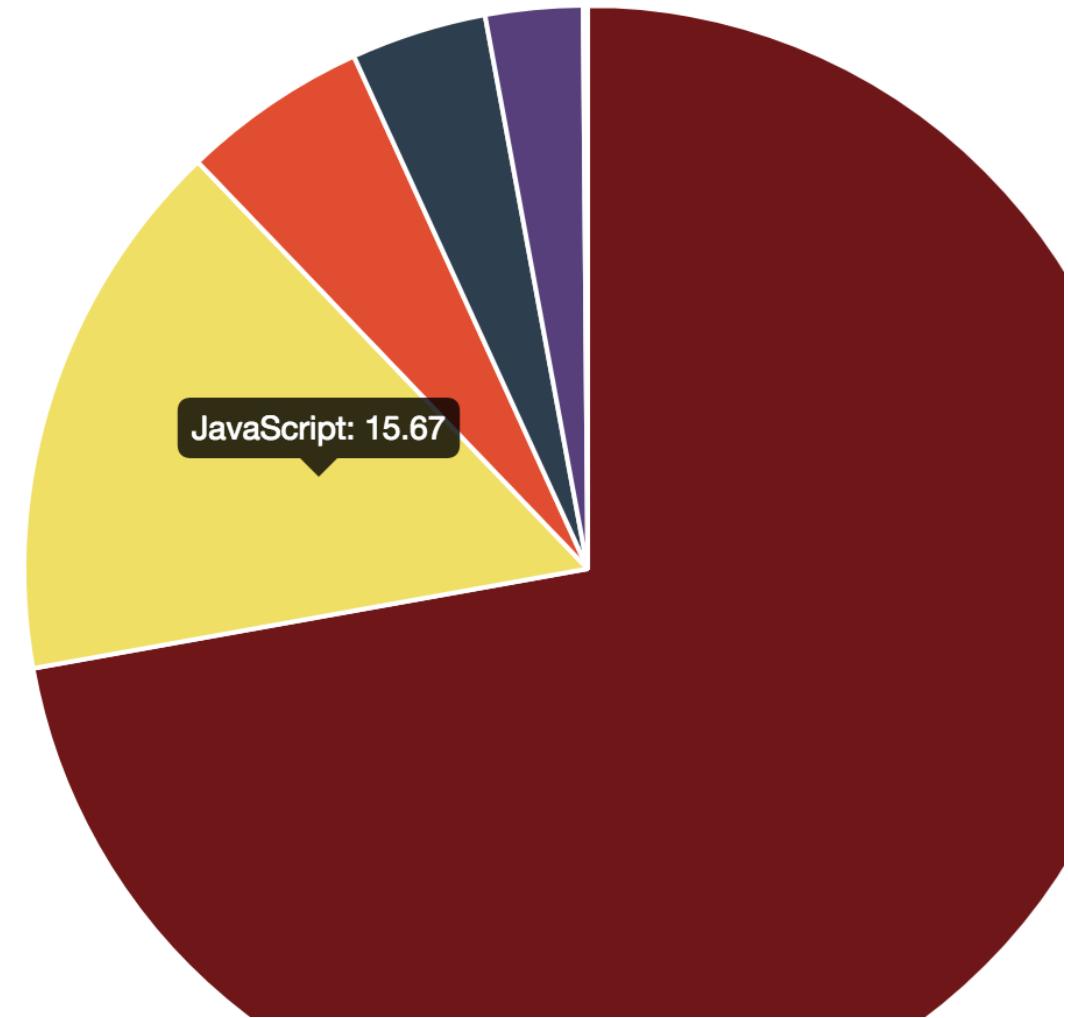


Features

- Among these tools are charting tools, which are used to get a better understanding of the composition of the repository:

Programming languages used in this repository

● Ruby	72.17 %
● JavaScript	15.67 %
● HTML	5.35 %
● Vue	3.89 %
● CSS	2.79 %
● Shell	0.13 %
● Clojure	0.0 %



Commit statistics for master Jun 01 - Jul 26

- Total: **2000 commits**
- Average per day: **35 commits**
- Authors: **204**

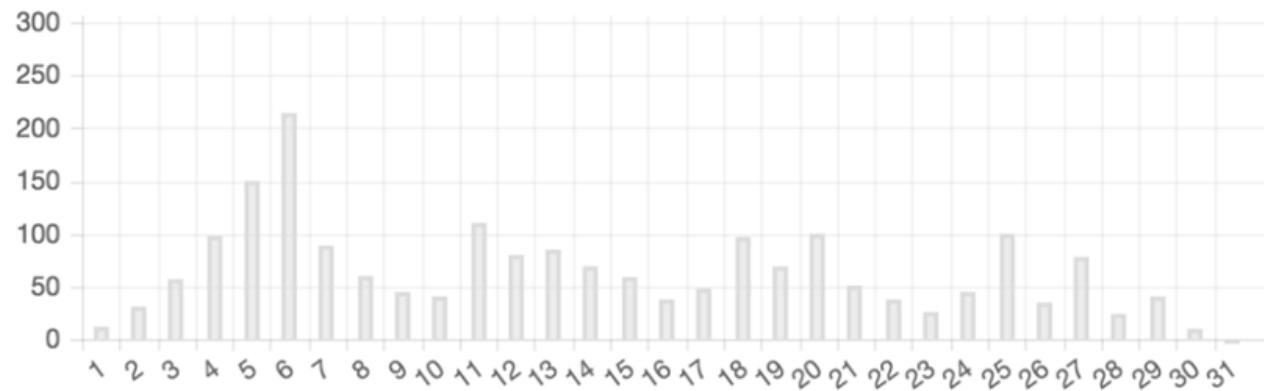
Features

- Alongside this is a breakdown of the frequencies of commits and activity:

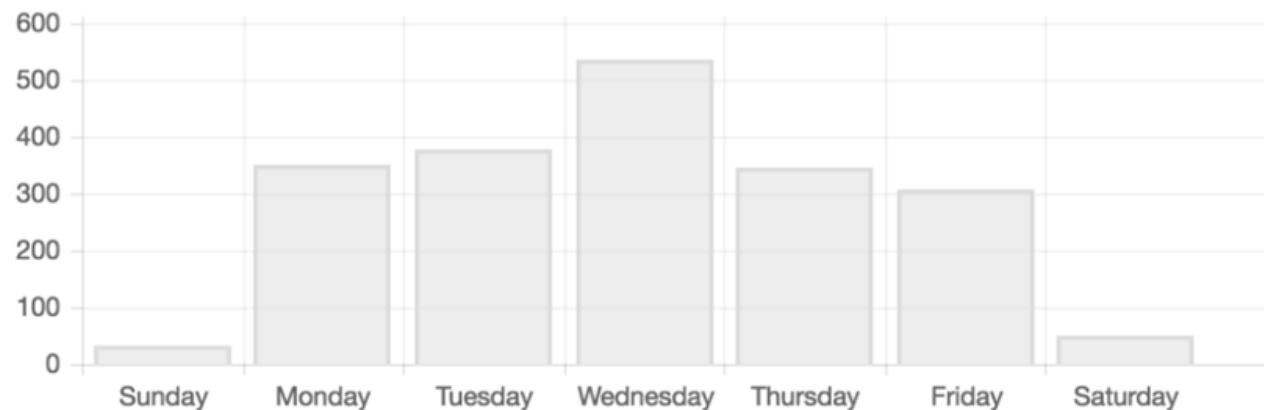
master

gitlab-ce

Commits per day of month



Commits per weekday



Features

- GitLab also provides a web interface where you can make changes to code and commit it straight from the browser:



The screenshot shows a GitLab interface for editing files. On the left, there's a sidebar with 'Edit' and a '+' button, followed by a 'Files' section containing a 'Docs' folder. Inside 'Docs', several files are listed: 11.rtf, 11_notes.rtf, 11_synopsis.txt, 12.rtf (which is selected), 29.rtf, 30.rtf, 35.rtf, 38.rtf, 39.png, 4.rtf, 42.rtf, 45.rtf, 46.rtf, 47.rtf, 48.rtf, 49.rtf, 50.rtf, and 51.rtf. At the bottom of the sidebar, there's a 'Commit...' button and a message '0 unstaged and 0 staged changes'. The main area displays the content of the selected file, 12.rtf, which contains RTF template code. The code includes various RTF commands like \f0\fs42, \fs31\fsmilli15750, and \pard\tx560\tx1120\tx1680\tx2240\tx2800\tx3360\tx3920\tx4480\tx5040\tx5600\tx6160\tx6720\sl288\slmult1\sb160\pardirnatural\qj. Lines 26 through 28 contain a note: 'When compiled (File > Compile), this project will generate a document in standard manuscript format with sub-headings inside the chapters.\'

```
1 {\rtf1\ansi\ansicpg1252\cocoartf1347\cocoasubrtf570
2 {\fonttbl\f0\fnil\fcharset0 Baskerville;\f1\fnil\fcharset0 CenturySchoolbook;\f2\fswiss\fcharset0 Helvetica;
3 \f3\fnil\fcharset0 Cochin;}
4 {\colortbl;\red255\green255\blue255;}
5 {\*\listtable{\list\listtemplateid1\listhybrid
6 {\listlevel\levelnfc23\levelfcn23\leveljc0\leveljc0\levelfollow0\levelstartat1\levelspace360\levelindent0{\*\levelmarker \
7 {disc}}{\leveltext\leveltemplateid1\'01\uc0\u8226 ;}{\levelnumbers;}\fi-360\li720\lin720 }{\listname ;}\listid1}
8 {\list\listtemplateid2\listhybrid
9 {\listlevel\levelnfc23\levelfcn23\leveljc0\leveljc0\levelfollow0\levelstartat1\levelspace360\levelindent0{\*\levelmarker \
10 {disc}}{\leveltext\leveltemplateid101\'01\uc0\u8226 ;}{\levelnumbers;}\fi-360\li720\lin720 }{\listname ;}\listid2}
11 {\list\listtemplateid3\listhybrid
12 {\listlevel\levelnfc23\levelfcn23\leveljc0\leveljc0\levelfollow0\levelstartat1\levelspace360\levelindent0{\*\levelmarker \
13 {disc}}{\leveltext\leveltemplateid201\'01\uc0\u8226 ;}{\levelnumbers;}\fi-360\li720\lin720 }{\listname ;}\listid3}
14 {\*\listoverridetable{\listoverride\listid1\listoverridecount0\ls1}{\listoverride\listid2\listoverridecount0\ls2}
15 {\listoverride\listid3\listoverridecount0\ls3}}
16 \pard\tx560\tx1120\tx1680\tx2240\tx2800\tx3360\tx3920\tx4480\tx5040\tx5600\tx6160\tx6720\sl288\slmult1\sb160\pardirnatural\qj
17 \f0\fs42 \cf0 N
18 \fs31\fsmilli15750 ON
19 \fs42 -F
20 \fs31\fsmilli15750 ICTION
21 \fs42
22 \fs31\fsmilli15750 WITH
23 \fs42 S
24 \fs31\fsmilli15750 UB
25 \fs42 -H
26 \fs31\fsmilli15750 EADS
27 \f1\fs28 \
28 \pard\tx560\tx1120\tx1680\tx2240\tx2800\tx3360\tx3920\tx4480\tx5040\tx5600\tx6160\tx6720\sl360\slmult1\pardirnatural\qj
29 \f2\fs24 \cf0 \
30 
31 \fs36 About This Template
32 \fs24 \
33 When compiled (File > Compile), this project will generate a document in standard manuscript format with sub-headings inside the
34 chapters.\
```

Features

- With things like epics, milestones, and cycle analytics, GitLab can help measure the effectiveness of your development process.

▼ Backlog 5552

Remove trigger.owner #36794
CI/CD breaking change technical debt

GitLab pages support for subgroups #30548
CI/CD customer devops:release
feature proposal pages subgroups

Support new issue creation by email without subaddressing #29951
P3 Plan S3 customer customer+ emails feature proposal issues reply by email

Collect usage data on issue boards #25288
Plan boards

Filter Merge Requests by target branch #33831 4
Community Contribution Create P3 S3 UX ready auto updated customer devops:create feature proposal merge requests potential proposal repository

regression 92

Reinstate option to force lowercase project/repository names on creation #1989
feature proposal regression

PostgreSQL DB Migration Error When Upgrading to 9.2.0 #32721
CI/CD auto updated awaiting feedback customer database regression

Error 500 loading merge requests due to `nil` parameter in Repository#merge_base_commit #32812
Next Patch Release Platform backend bug regression repository reproduced on GitLab.com

Rspec feature tests have useless javascript stack traces #34012
backstage blocked regression test webpack

New issue from failed job not always workina #35874

direction 151

JUnit XML Test Summary In MR widget #45318
CI/CD Deliverable In dev Product Vision 2018 UX ready backend customer devops:verify direction feature proposal frontend merge requests

Pipeline view of environments #28698 9
CI/CD Deliverable deploy devops:release direction feature proposal frontend meta missed-deliverable

Method to update managed apps in Kubernetes #42686
CI/CD UX backend devops:configure direction feature proposal frontend kubernetes

Track label add/remove events for issues/mrs/epics #47993 5
Deliverable In dev Plan backend direction epics feature proposal issues labels merge requests

Features

- It also includes the necessary tools for code review prior to merging branches to ensure that all work is up to scratch:

Showing 2 changed files ▾ with 6 additions and 0 deletions

Hide whitespace changes Inline Side-by-side

...
270	270	@@ -270,6 +270,7 @@
271	271	.block {
272	272	width: 100%;
273	+	word-break: break-word;
273	274	
274	275	&:last-child {
275	276	border-bottom: 1px solid \$border-gray-normal;
...	...	

gist Edit View file @ cff585df

Features

- Automated testing tools and pipelines are also included to help make sure that code is working perfectly before it's merged back in or released:

passed Pipeline #26392237 triggered 1 hour ago by Winnie Hellmann

Enable no-console ESLint rule for tests

93 jobs from [winh-lint-console-tests](#) in 62 minutes 18 seconds (queued for 9 seconds)

7a870e40

SAST detected 59 vulnerabilities

Dependency scanning detected 133 vulnerabilities

Pipeline Jobs 93 Security report 192

Build

Prepare

Test

Post-test

package-and-qa

review-docs-d...

compile-assets

retrieve-tests-d...

setup-test-env

codequality

danger-review

db:check-sche...

db:migrate:res...

db:migrate:res...

coverage

flaky-examples...

lint:javascript:r...

Self-managed versus Software as a Service (SaaS)

- GitLab can be used in one of two ways: either self-managed, where you host your own instance of GitLab Community Edition/Enterprise Edition
- Using the online platform GitLab.com, which comes as a paid or free Software as a Service (SaaS) model.

Free versus paid

- Lastly, there are multiple tiers of GitLab for both the self-managed and SaaS versions.
- Please note that both versions can be used for free and provide all of the main features that you'd expect (git hosting, code review, issue management, testing, and deployment).
- The added tiers provide extra features that are available at different levels of pricing on a per-user, per-month basis.

Summary

- So far, we've discovered what version control is: a method of tracking revisions of work, of creating alternate test branches, and working collaboratively.
- We know that git is a form of version control system that specializes in working in a distributed network and that GitLab is a platform that is based on git but with a lot of powerful features.

2. GitLab Flow



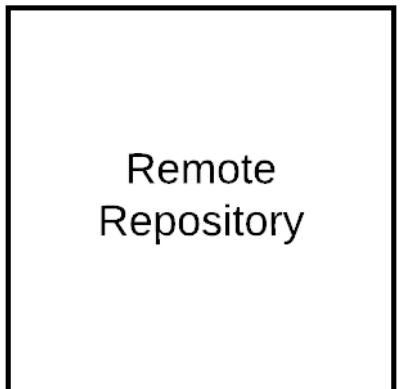
GitLab Flow

In this lesson, we'll explore the following topics:

- How to use Git
- The GitFlow for branching and merging
- The GitLab flow (recommended by GitLab)
- The GitLab flow in comparison to the GitHub flow

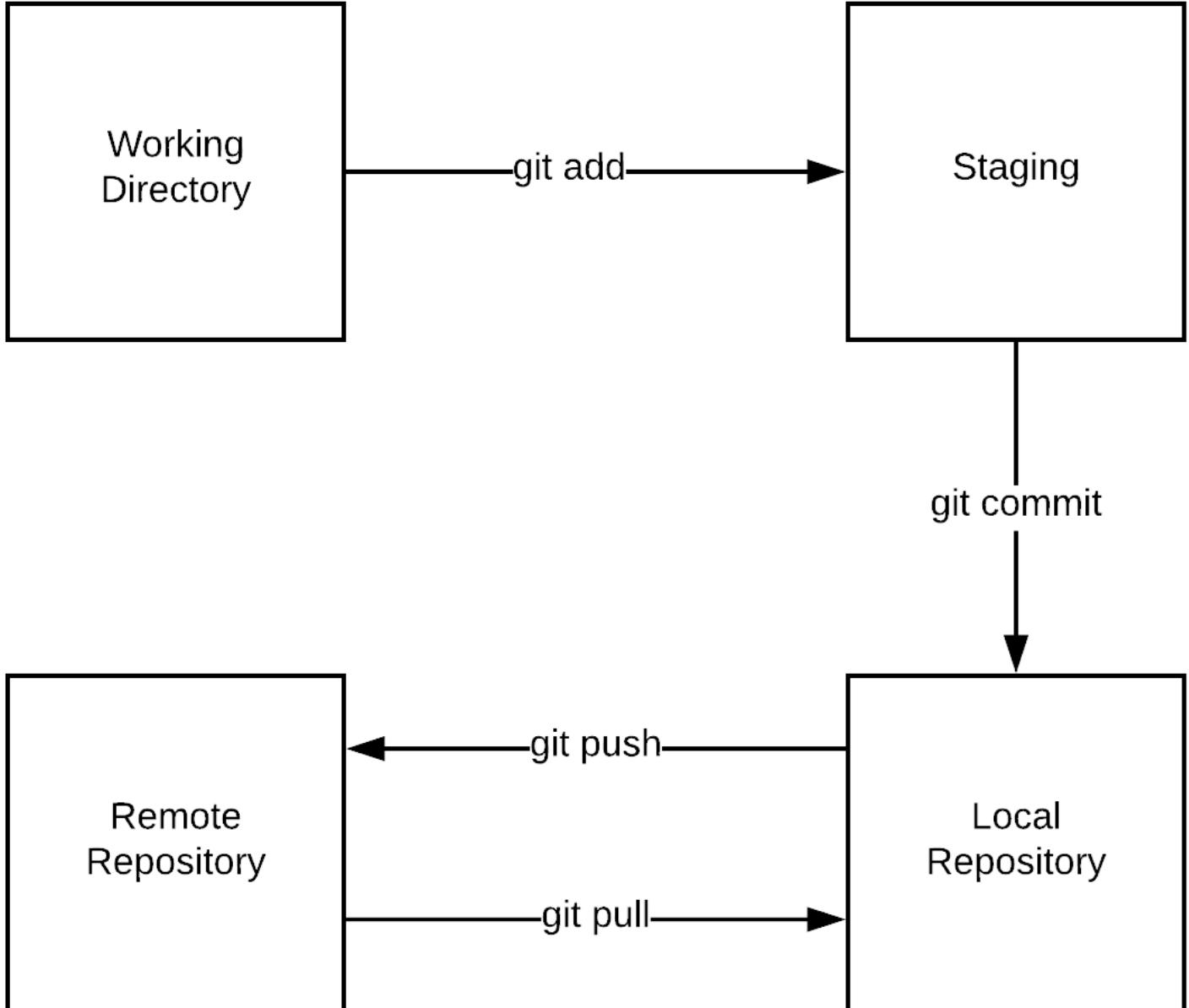
Using Git

- One thing to recognize is the life cycle of Git work and the stages it can go through.
- There are four separate parts: your working directory, the index or staging area, committing to the local repository, and pushing to remote repositories:



Using Git

- Overall, it seems like a pretty complicated workflow, which is why I've created this handy graph to illustrate the basic operations that you can perform and how they relate to these stages:

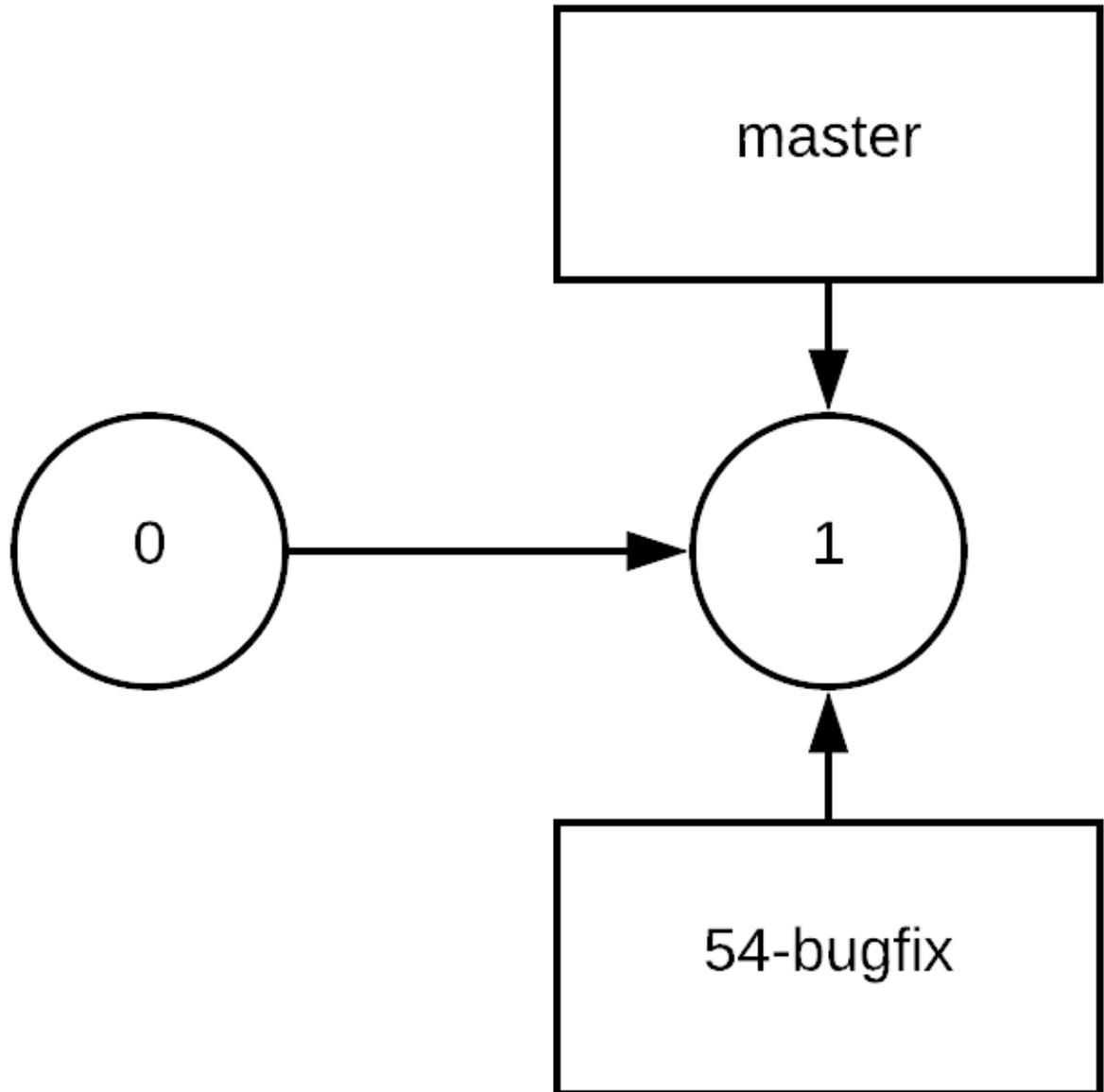


Git commands

- I've introduced a few Git commands while discussing how Gitworks, but haven't actually explained properly how to use them, so let's do a very brief introduction to the basic usage of Git.
- The first command you'll need when starting a new project is `git init`. This initializes a new Git repository, and you'll notice that it creates a `.git` folder so that you can start your project (this is hidden by default on Unix-based and Windows operating systems).

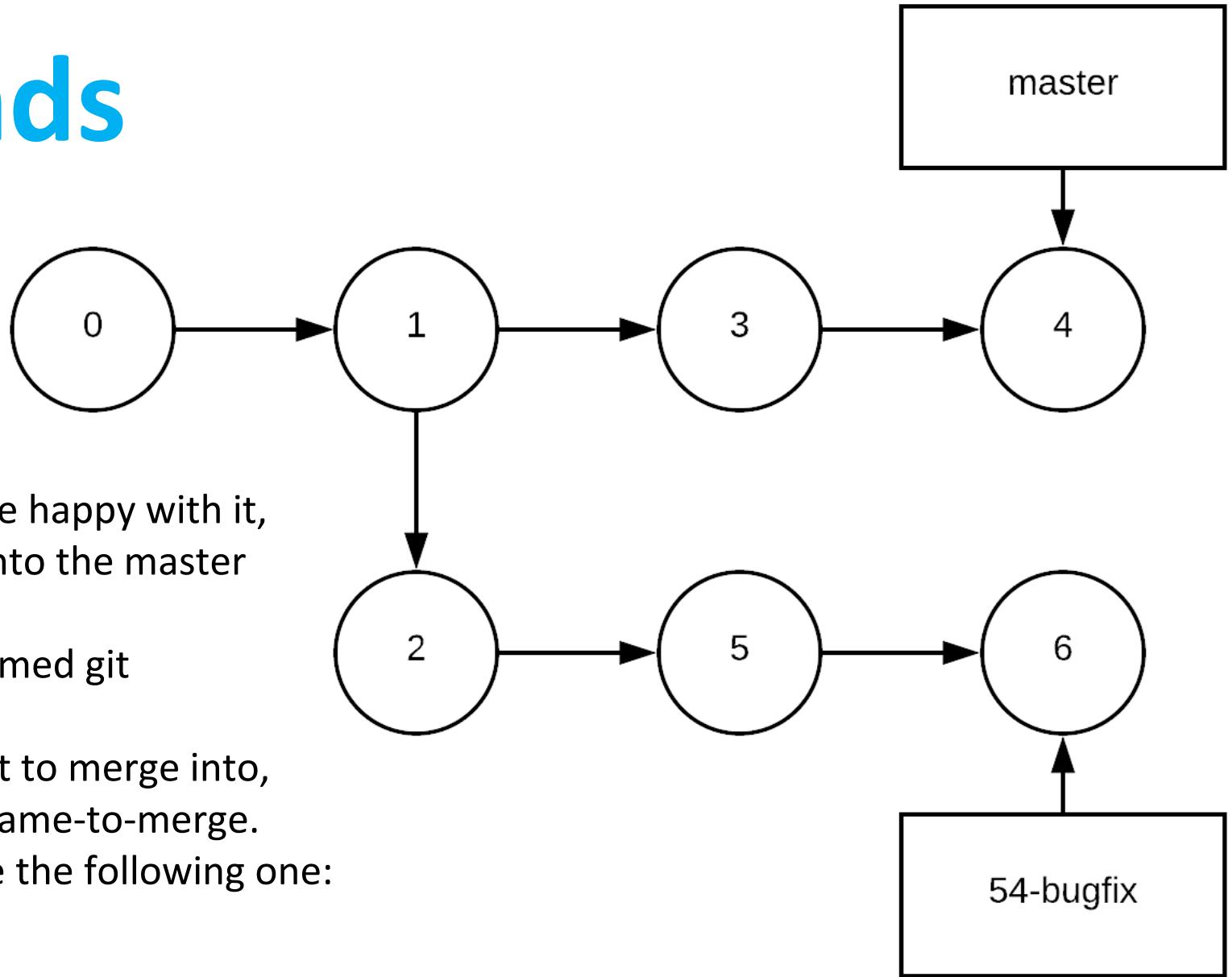
Git commands

- Behind the scenes, a Git branch is a reference to a particular commit, and adding new commits moves the branch pointer forward, but forks off from the main (master) branch.
- In this way, they are lightweight and still connected to the entire history of the local repository:



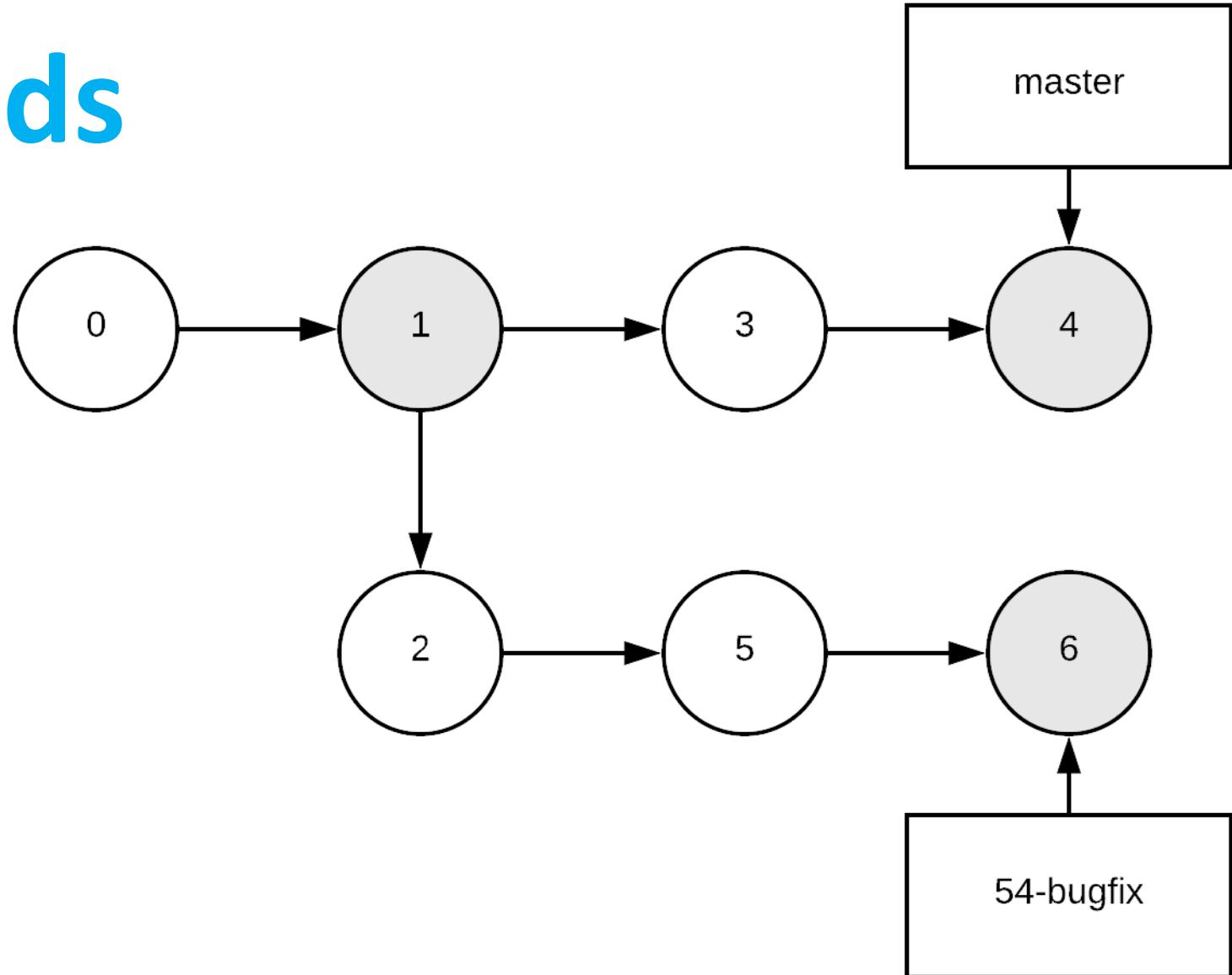
Git commands

- If you've done some work and are happy with it, you may want to merge it back into the master branch of the code.
- You can do this with the aptly named git merge command.
- You swap to the branch you want to merge into, and then run git merge branch-name-to-merge.
- Let's say you have a situation like the following one:



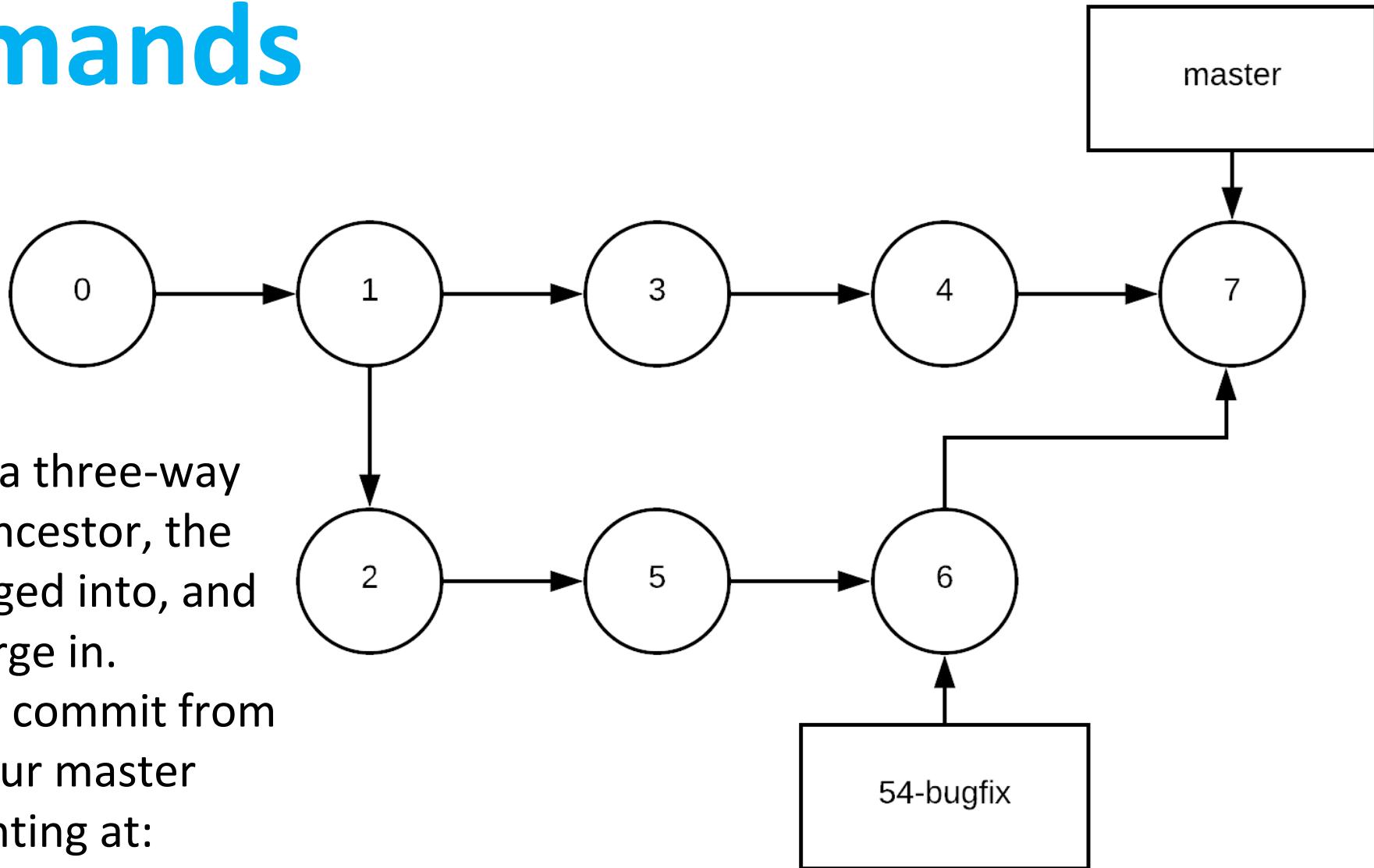
Git commands

- When you're on the master branch and run git merge 54-bugfix, behind the scenes, Git grabs those two commits and then finds their closest common ancestor:



Git commands

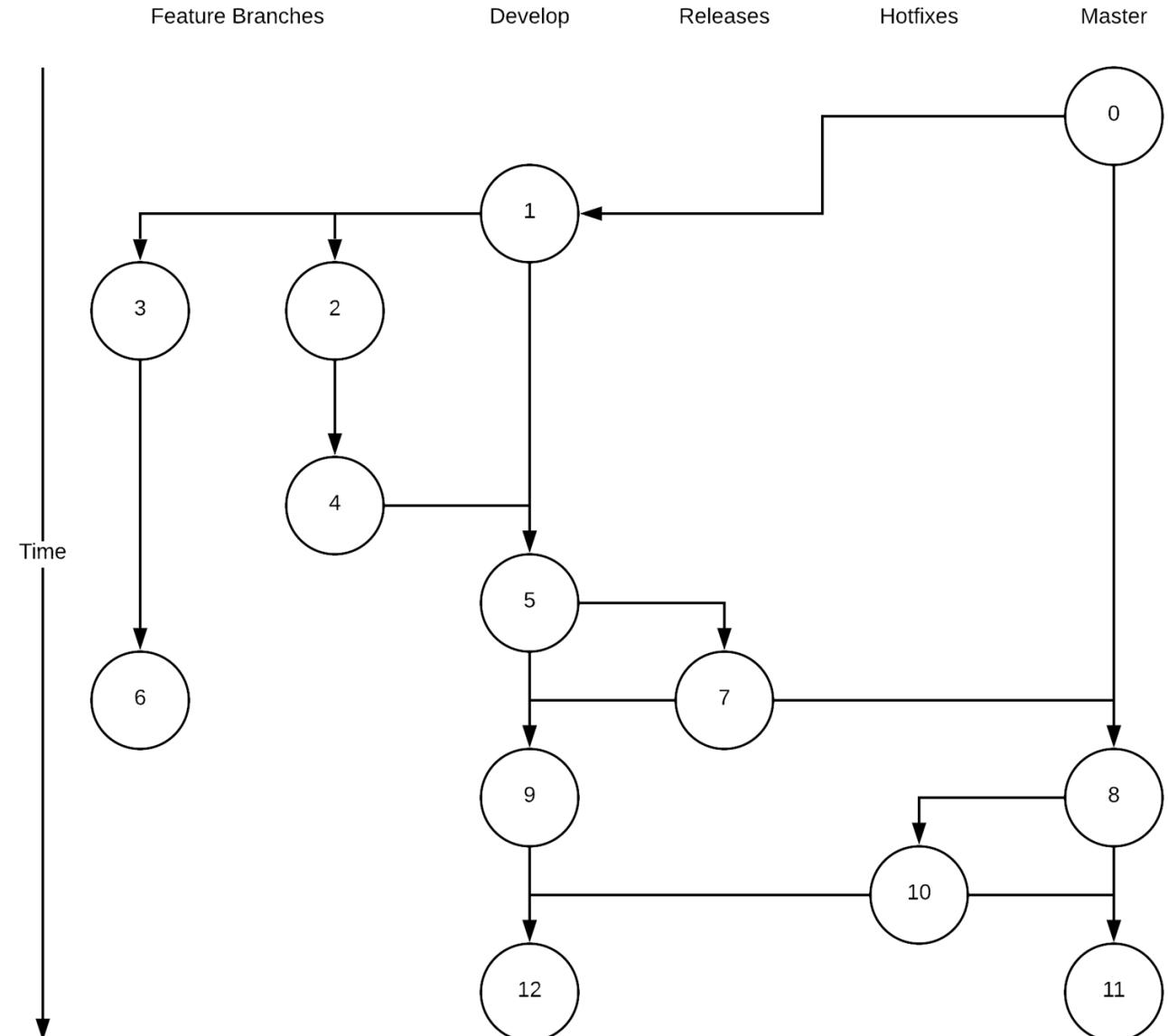
- Git then attempts a three-way merge using the ancestor, the commit to be merged into, and the commit to merge in.
- This creates a new commit from all three, which your master branch is now pointing at:



GitFlow

- Git flow is a semi-standardized workflow for dealing with projects in Git.
- It can be daunting, but the rules involve creating a stringent practice for when branches should be committed to or merged in order to prevent deploying buggy code or releases.

A diagram of it is as follows:



Master

The master branch is the original branch where all code is branched from.

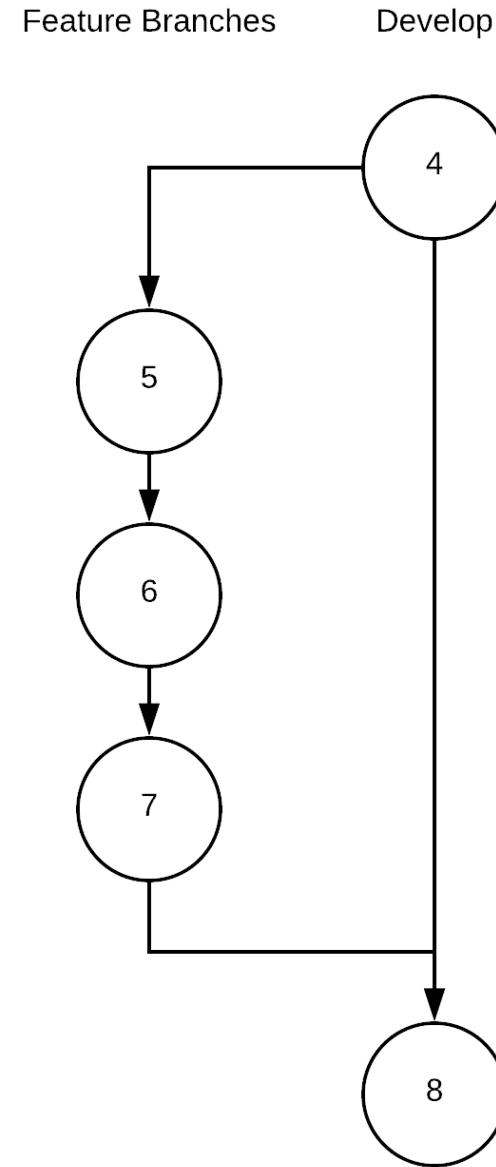
- However, it is never committed to directly; it is only ever merged into.
- The master branch is then used as the deployed branch, or the released branch if you have a product that is shipped. You can tag each merge with a version number to help identify these releases or deployed versions.

Develop

- The develop branch is the other continuously existing branch in the Git flow model, along with the master branch.
- All other branches are considered temporary and can be deleted after work is finished on them and they've been merged.

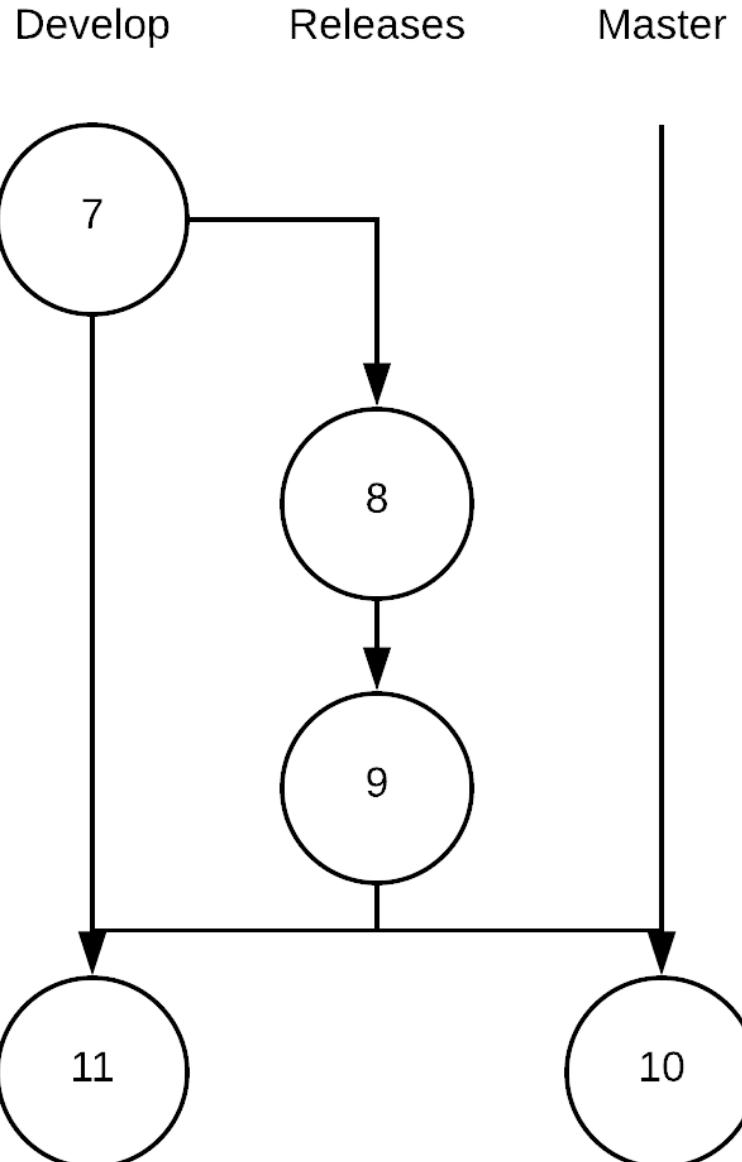
Features

- Unlike the develop or master branches, feature branches only need to last until the feature is complete and merged into develop, after which you can delete the branch since it should no longer be needed:



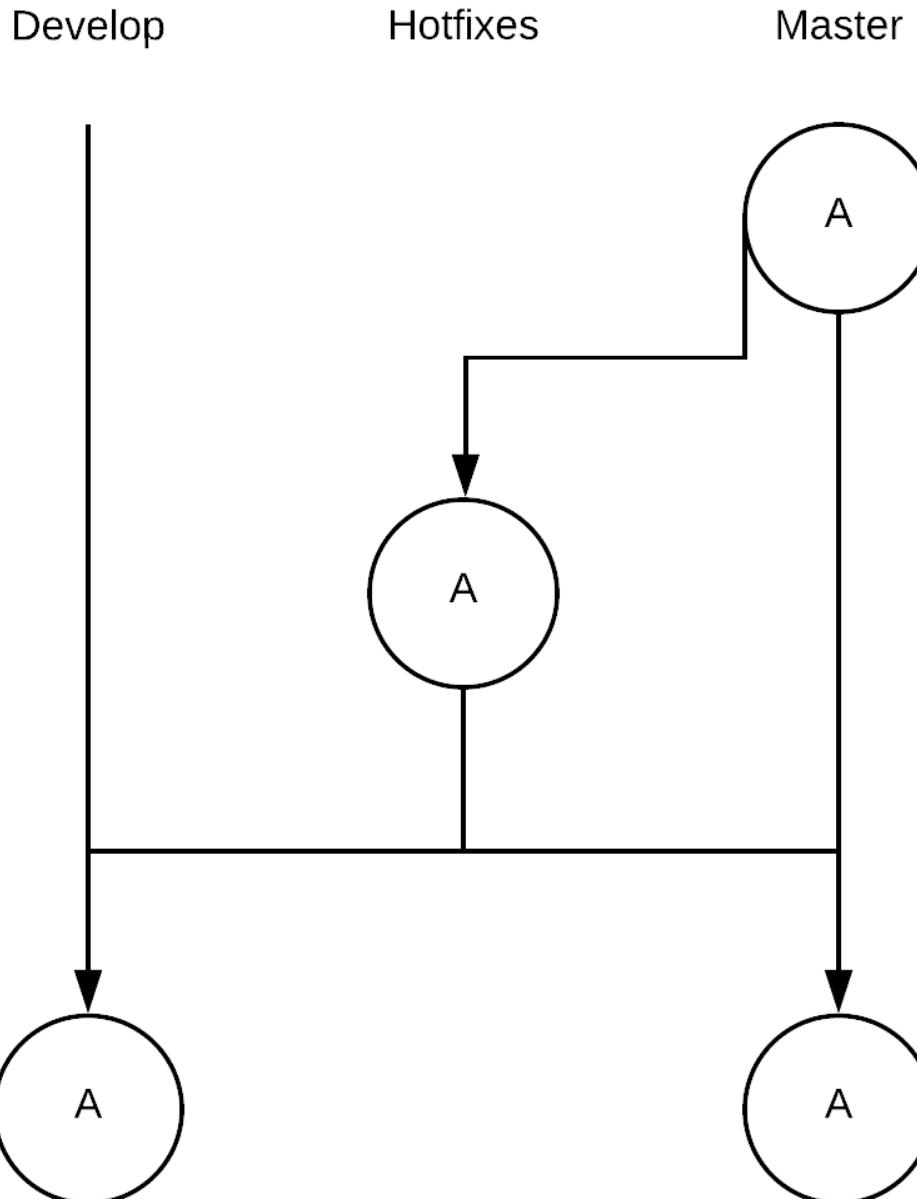
Releases

- If everything is ready, the branch can be merged directly into the master and tagged appropriately as a new release (with either a major or minor version number, depending on your scheme).
- Once this is done, the branch should be remerged into develop once more to ensure that all of the changes are back into the main development stream.
- You can then safely delete this branch since it is no longer required:



Hotfixes

- After you've merged into master, you also need to merge this into develop so that develop has the latest patches and fixes and you don't conflict in future releases.
- The hotfix can then be safely deleted as it is no longer required:

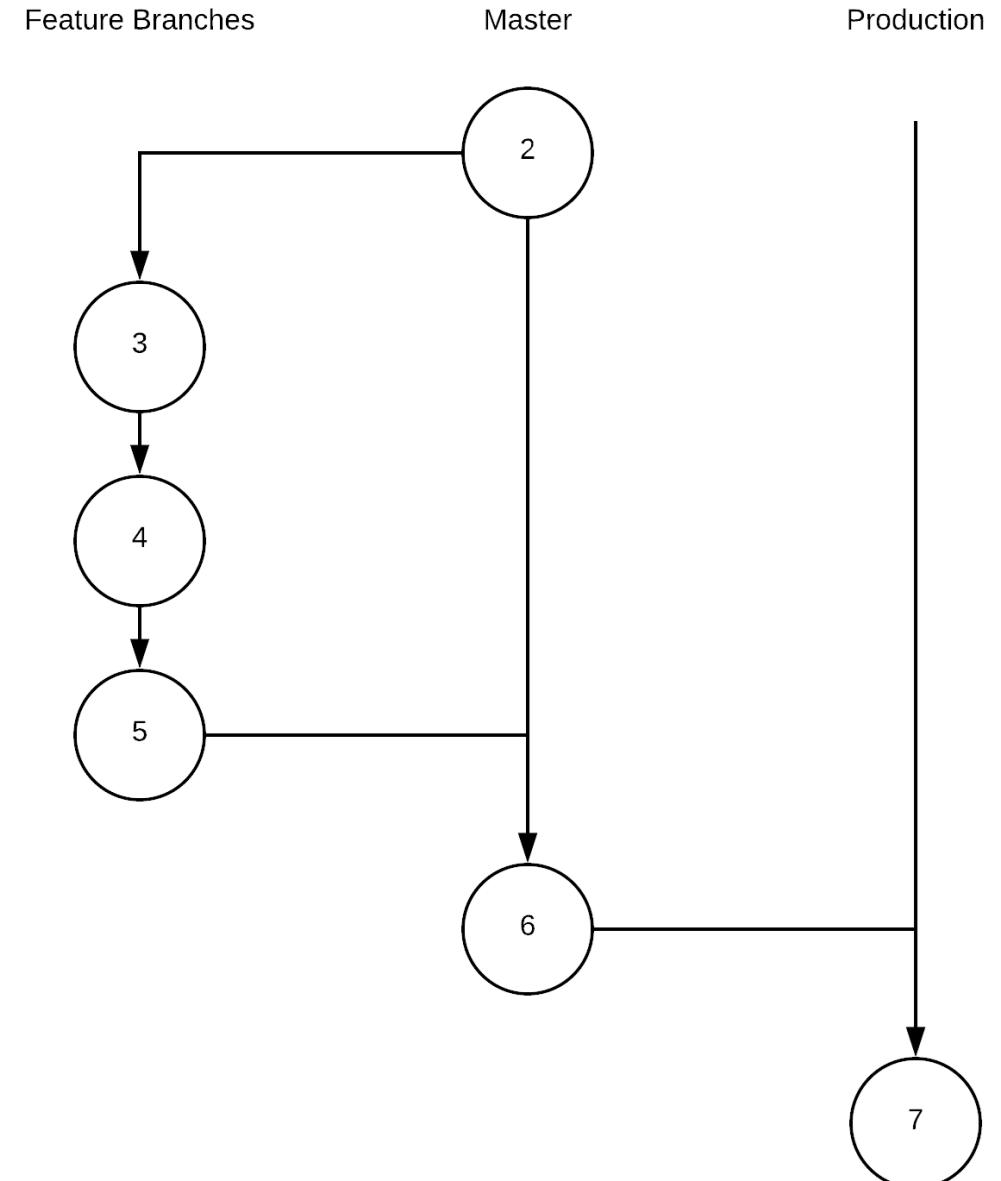


GitLab Flow

- So far, we've explored GitFlow for collaborative project management, but it can be a complex one that doesn't suit all needs.
- There are alternatives, though, and one of these is posited by GitLab and thus known as GitLab Flow.
- GitLab Flow is actually a collection of different branching strategies that can be used depending on your environments and needs.

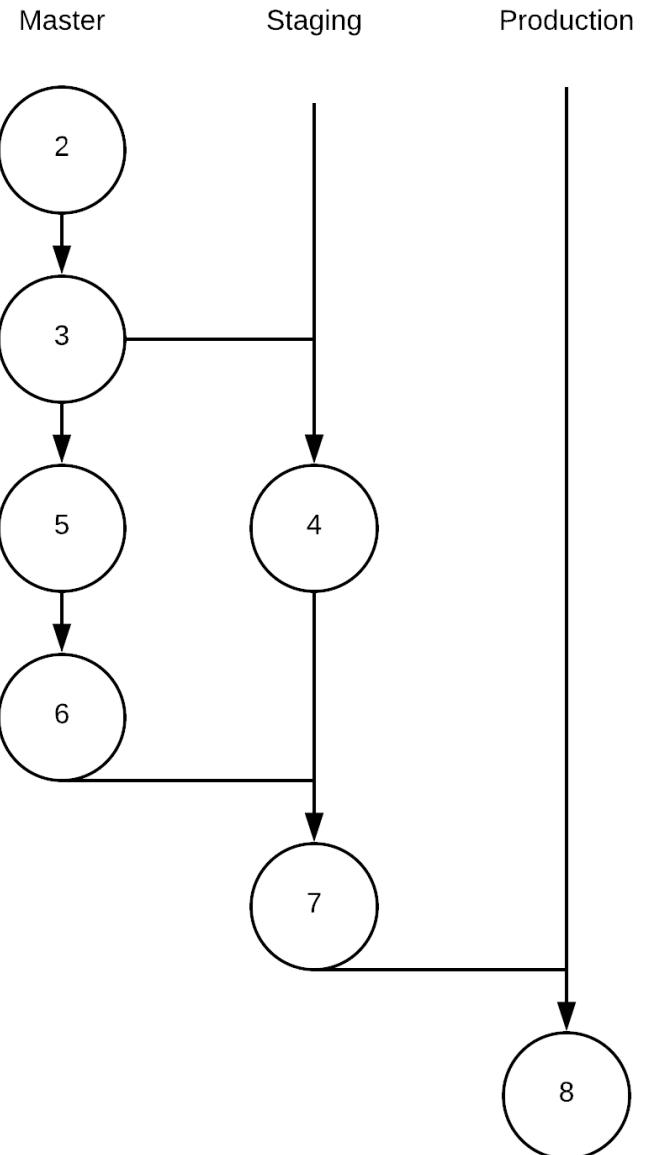
Production branch

- While some development environments can be deployed as soon as commits are merged in, some projects aren't set up like that.
- A good example is apps that need to be approved by the app store and thus aren't on the same agile, continuous release schedule as the code in the master branch, or platforms that can only be deployed during certain late night/early morning windows so as not to affect the users of the systems:



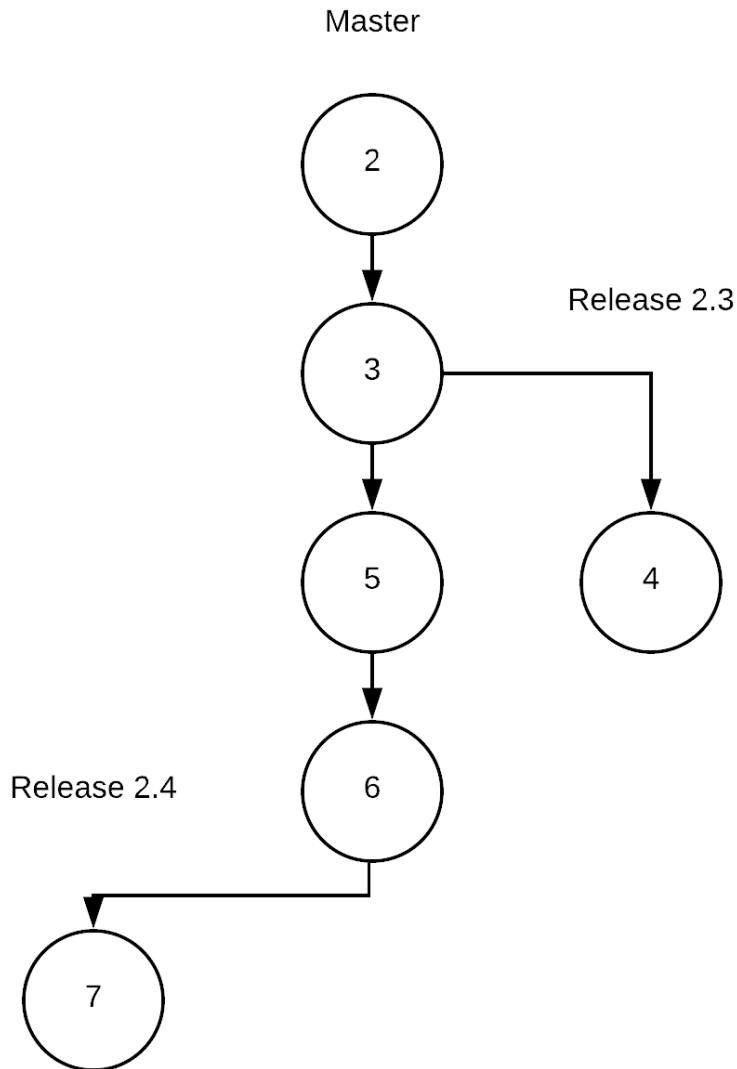
Staging branch

- The production branch workflow is good, but sometimes you have multiple environments and your QA team might want to test stuff a bit further before it gets released.
- For example, you might have a staging or pre-production environment that you want to deploy to and ensure that everything is working before you deploy to production.
- In that case, you can use a workflow like the following:



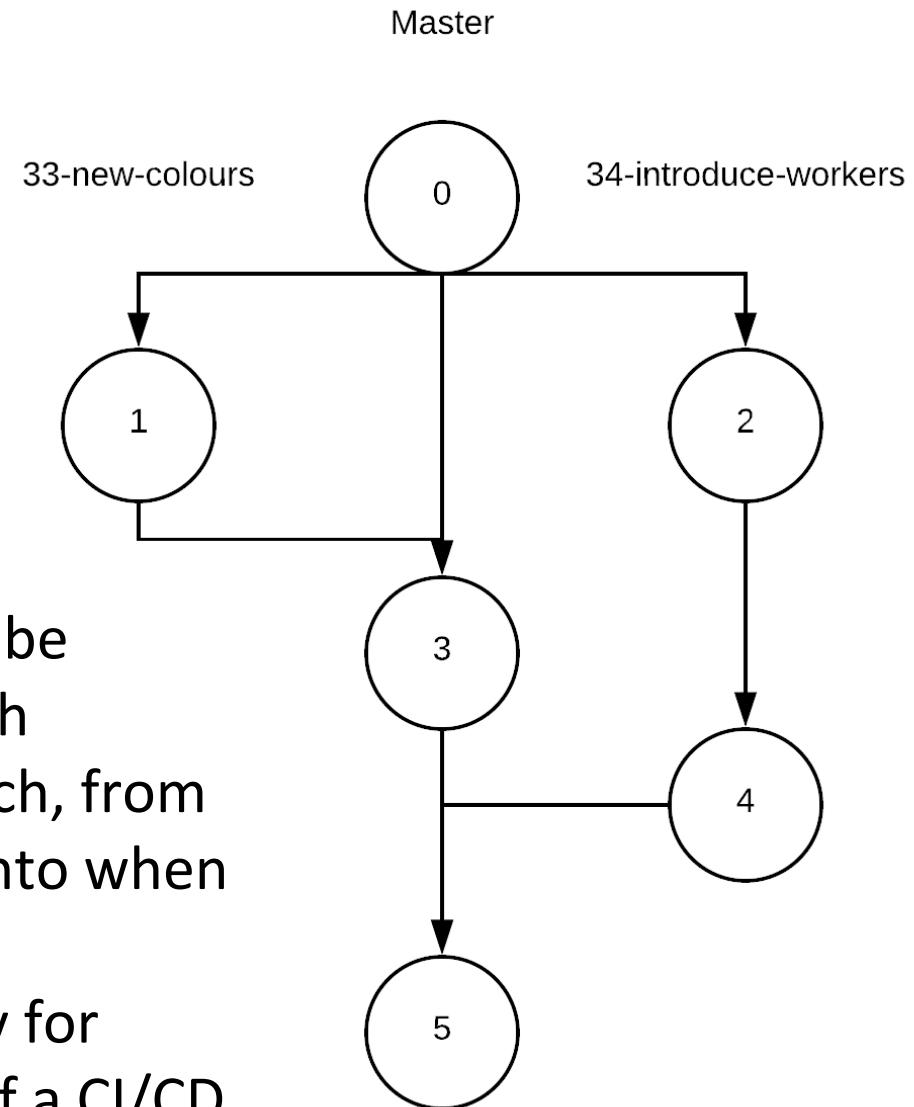
Release branch

- The last flow suggested by the GitLab team is the release branch flow.
- A handy diagram of it is included as follows:



Differences from GitHub flow

- If you've previously worked with GitHub, you might be familiar with the GitHub flow, a rigid workflow which recommends only having a continuous master branch, from which features are branched off and then merged into when created.
- Any code in the master branch is theoretically ready for deployment and continuous release with the help of a CI/CD platform:



Summary

- In this lesson, we've looked at how to apply Git to your software development process and we've covered most of the basic commands, including add, commit, push, pull, branch, and merge.
- We also looked at GitFlow, a recommended workflow for Git software projects, and how to handle branching and merging correctly.

Introducing the GitLab Architecture

In this lesson, we will be covering the following topics:

- The origins of GitLab
- GitLab CE or EE
- The core components of GitLab
- GitLab CI
- GitLab Runners
- Cloud native

The origins of GitLab

- The story began in 2011, when Dimitri Zaporozhets, a web programmer from Ukraine, was faced with a common problem.
- He wanted to switch to Git for version management and GitHub to collaborate, but that was not allowed in his company.
- He needed a tool that did not hinder him in developing code and was easy to use.

The origins of GitLab

- After this initiative, the project grew enormously:

Date	Fact
2011	Sytze Sybrandij, the future CEO of GitLab, is impressed by the GitLab project and code, and offers Zaporozhets the opportunity to try to commercialize it via https://about.gitlab.com/ .
2012	GitLab was announced to a broader audience via Hacker News (https://news.ycombinator.com/item?id=4428278).
2013	Dimitri Zaporozhets decides to work full-time on GitLab and joins the company.
2015	GitLab becomes part of the Y Combinator class and received VC funding that year.
2018	GitLab receives another \$100 million of VC funding and is valued at \$1 billion.
2019	The GitLab company employs over 600 employees.

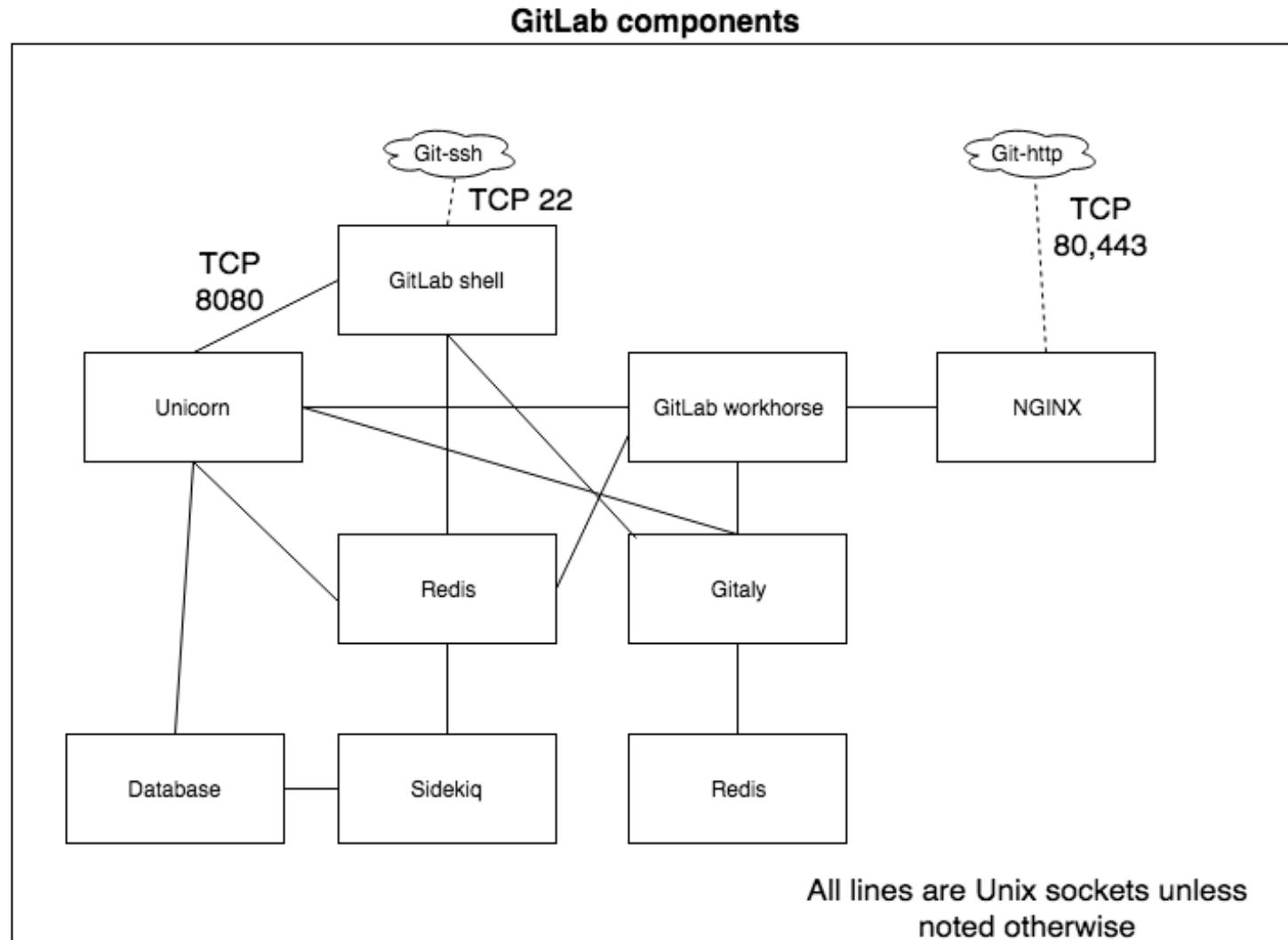
Exploring GitLab editions – CE and EE

- The core of the GitLab software is called the CE.
- It is distributed under the MIT license, which is a permissive free software license created at the Massachusetts Institute of Technology.
- You are allowed to modify the software and use it in your creations.

Exploring GitLab editions – CE and EE

Version	Features (short list)
Starter	Everything on core GitLab CE: <ul style="list-style-type: none">• CI/CD• Project Issue Board• Mattermost integrations• Time tracking• GitLab pages
Premium	More enterprise features such as the following: <ul style="list-style-type: none">• Maven and NPM repository functionality• Protected environments• Burndown charts• Multiple LDAP servers and Active Directory support
Ultimate	All options, including the following: <ul style="list-style-type: none">• All security scanning tools• Epics• Free guest users• Web terminal for the web IDE

The core system components of GitLab



Summary

- In this lesson, we have learned about the people and the organization behind GitLab.
- Starting from the beginning, we have shown you how the project has developed over the years.
- We went through the core components of GitLab.

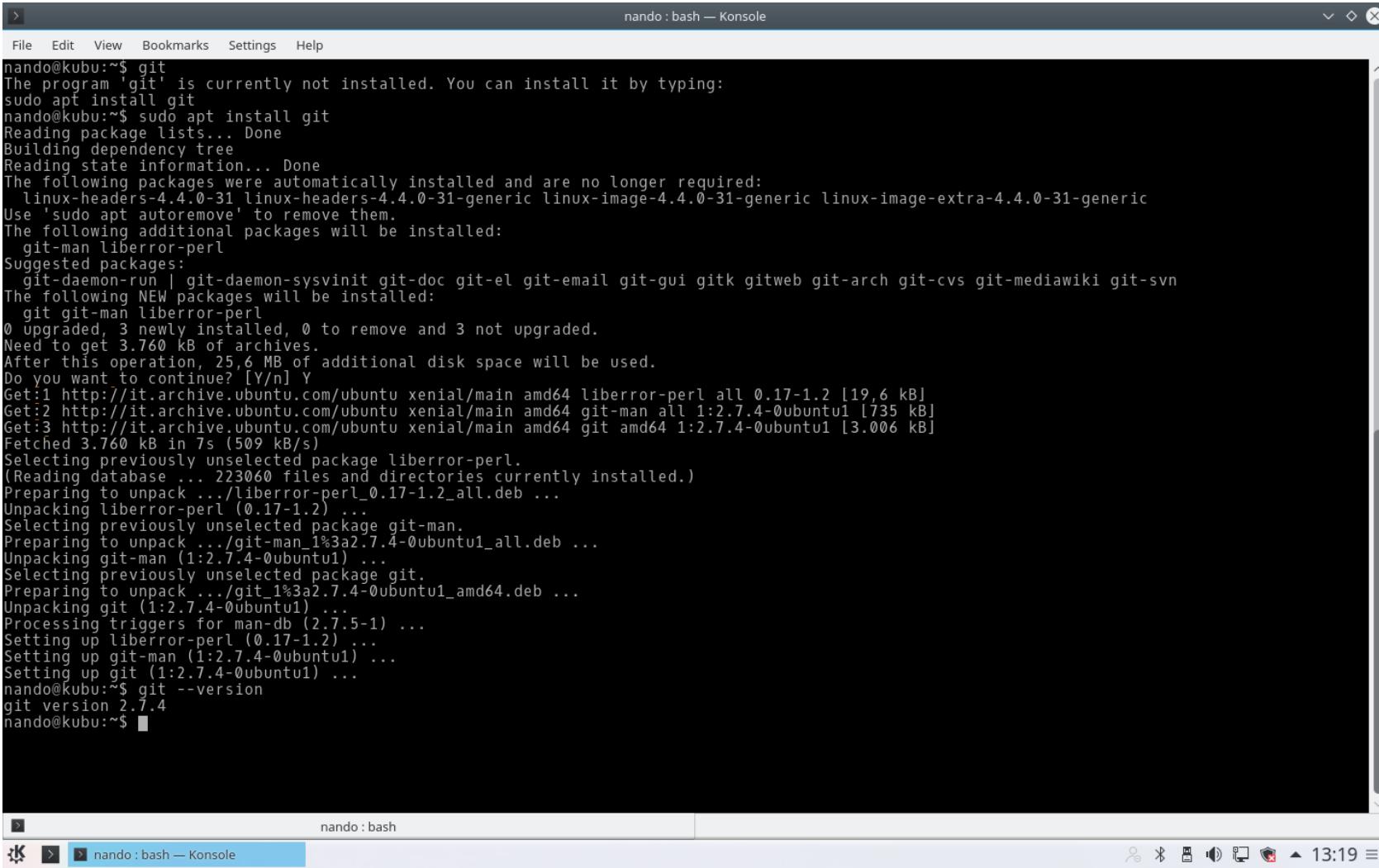
Getting Started with Git

- In this first lesson, we will start at the very beginning, assuming that you do not have Git on your machine.
- This course is intended for developers who have never used Git or only used it a little bit, but who are scared to throw themselves headlong into it.
- If you have never installed Git, this is your lesson.
- If you already have a working Git box, you can quickly read through it to check whether everything is alright.

Installing Git

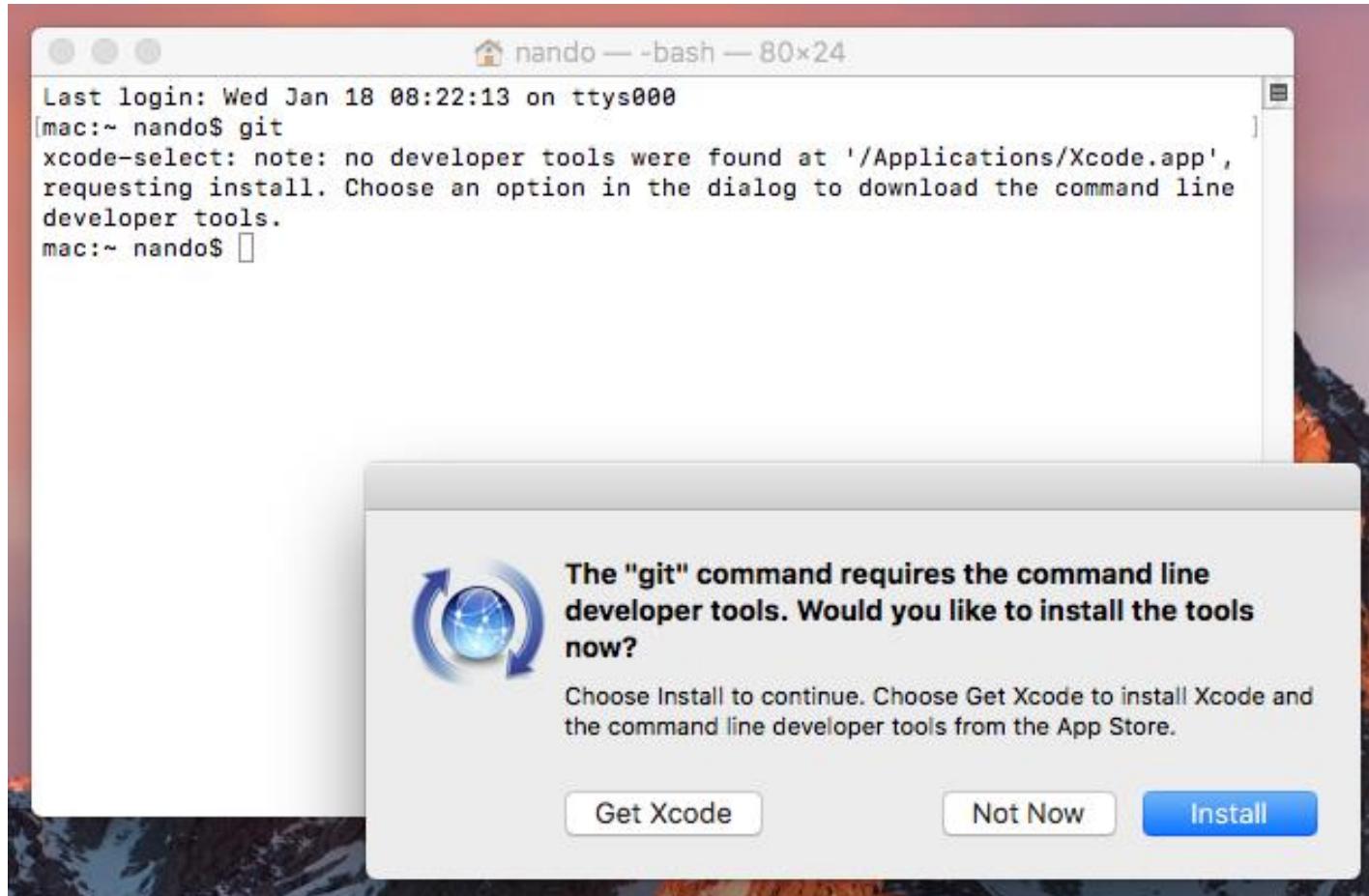
- Git is open source software.
- You can download it for free from <http://git-scm.com>, where you will find a package for all the most common environments (GNU-Linux, macOS and Windows).
- At the time of writing this course, the latest version of Git is 2.11.0.

Installing Git on GNU-Linux



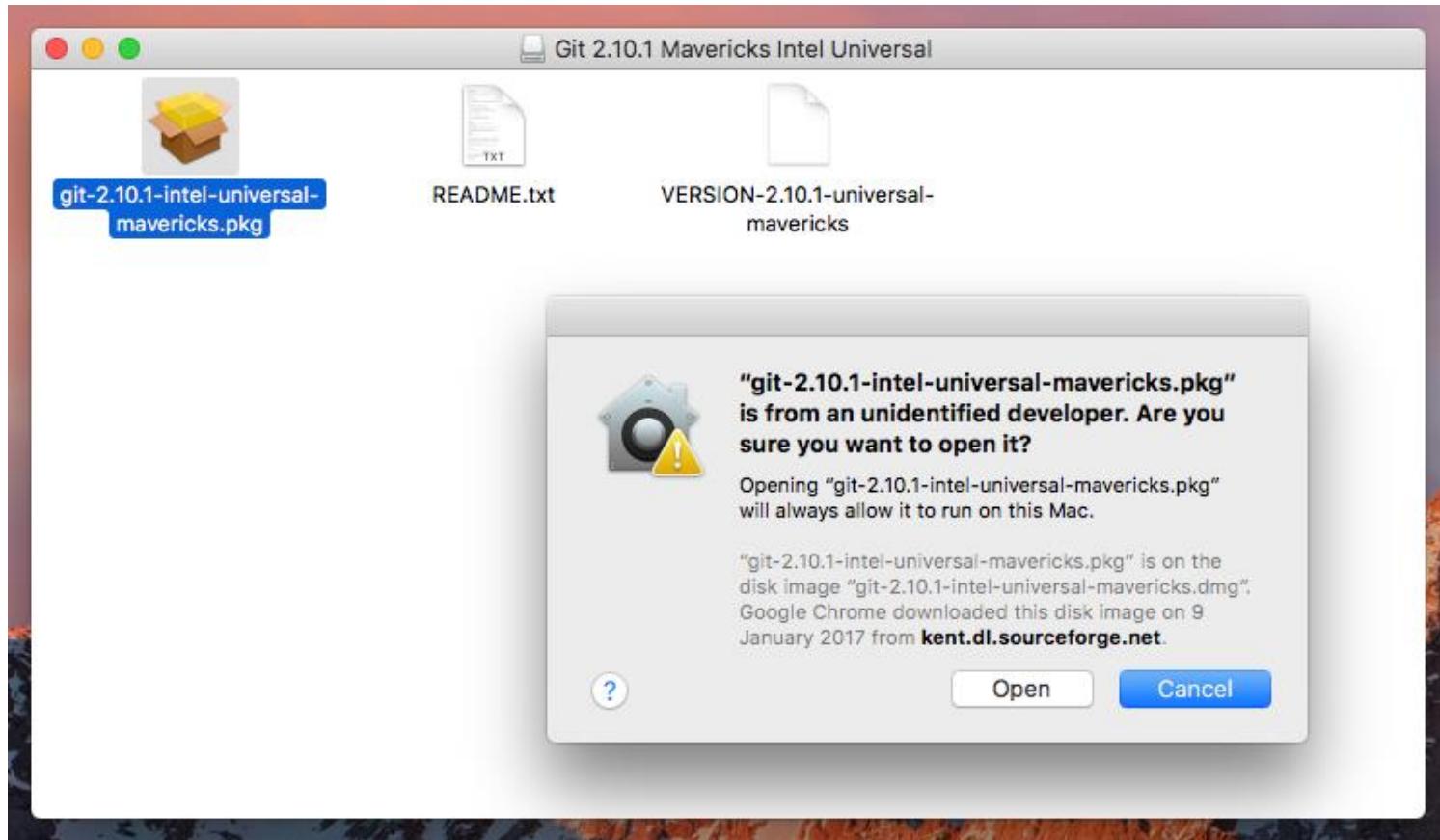
```
nando@kubu:~$ git
The program 'git' is currently not installed. You can install it by typing:
sudo apt install git
nando@kubu:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.4.0-31 linux-headers-4.4.0-31-generic linux-image-4.4.0-31-generic linux-image-extra-4.4.0-31-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  git-man liberror-perl
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-arch git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  git git-man liberror-perl
0 upgraded, 3 newly installed, 0 to remove and 3 not upgraded.
Need to get 3.760 kB of archives.
After this operation, 25,6 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://it.archive.ubuntu.com/ubuntu xenial/main amd64 liberror-perl all 0.17-1.2 [19,6 kB]
Get:2 http://it.archive.ubuntu.com/ubuntu xenial/main amd64 git-man all 1:2.7.4-0ubuntu1 [735 kB]
Get:3 http://it.archive.ubuntu.com/ubuntu xenial/main amd64 git amd64 1:2.7.4-0ubuntu1 [3.006 kB]
Fetched 3.760 kB in 7 s (509 kB/s)
Selecting previously unselected package liberror-perl.
(Reading database ... 223060 files and directories currently installed.)
Preparing to unpack .../liberror-perl_0.17-1.2_all.deb ...
Unpacking liberror-perl (0.17-1.2) ...
Selecting previously unselected package git-man.
Preparing to unpack .../git-man_1%3a2.7.4-0ubuntu1_all.deb ...
Unpacking git-man (1:2.7.4-0ubuntu1) ...
Selecting previously unselected package git.
Preparing to unpack .../git_1%3a2.7.4-0ubuntu1_amd64.deb ...
Unpacking git (1:2.7.4-0ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up liberror-perl (0.17-1.2) ...
Setting up git-man (1:2.7.4-0ubuntu1) ...
Setting up git (1:2.7.4-0ubuntu1) ...
nando@kubu:~$ git --version
git version 2.7.4
nando@kubu:~$ █
```

Installing Git on macOS



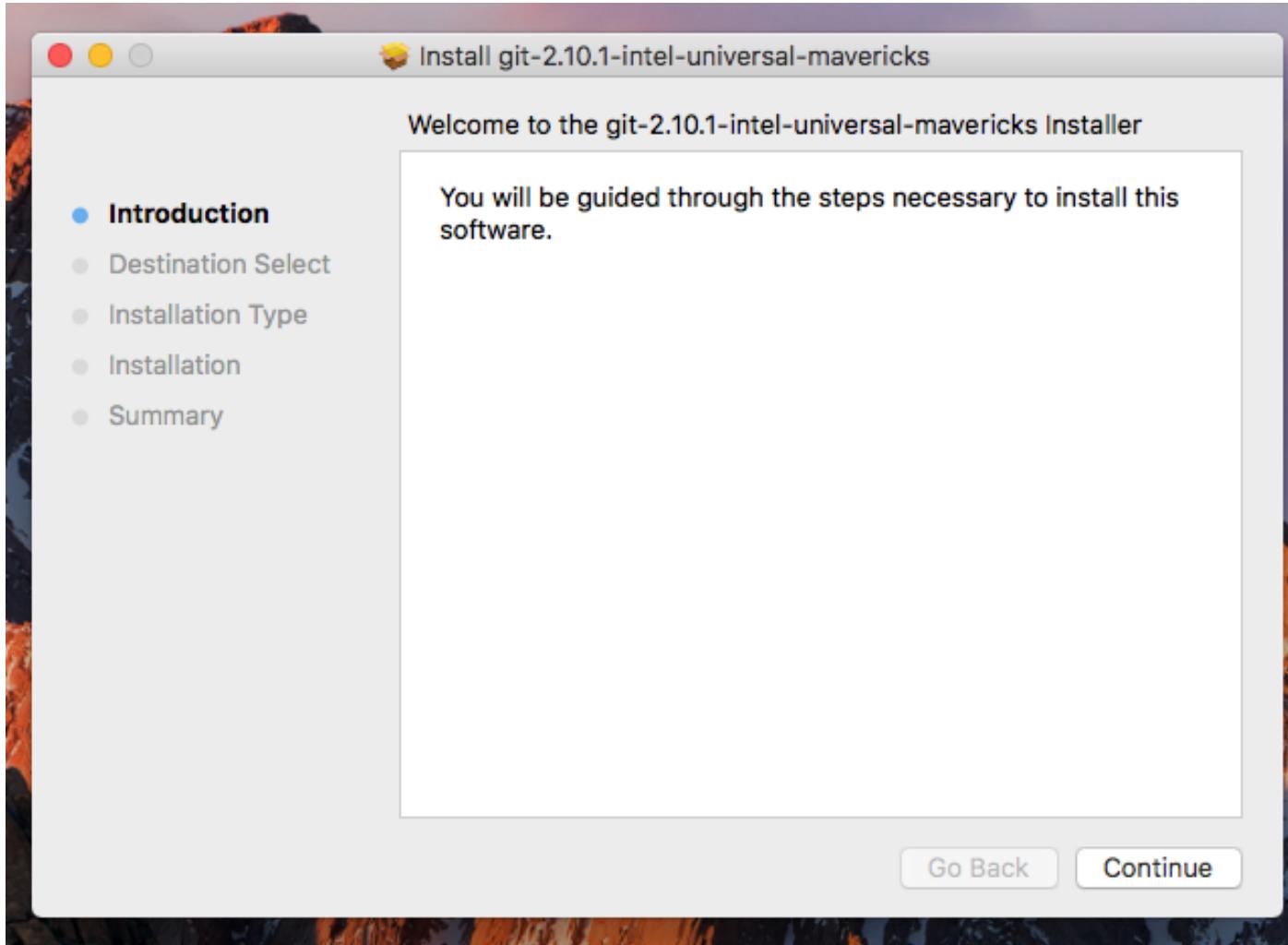
Clicking on the Install button will fire the installation process.

Installing Git on macOS



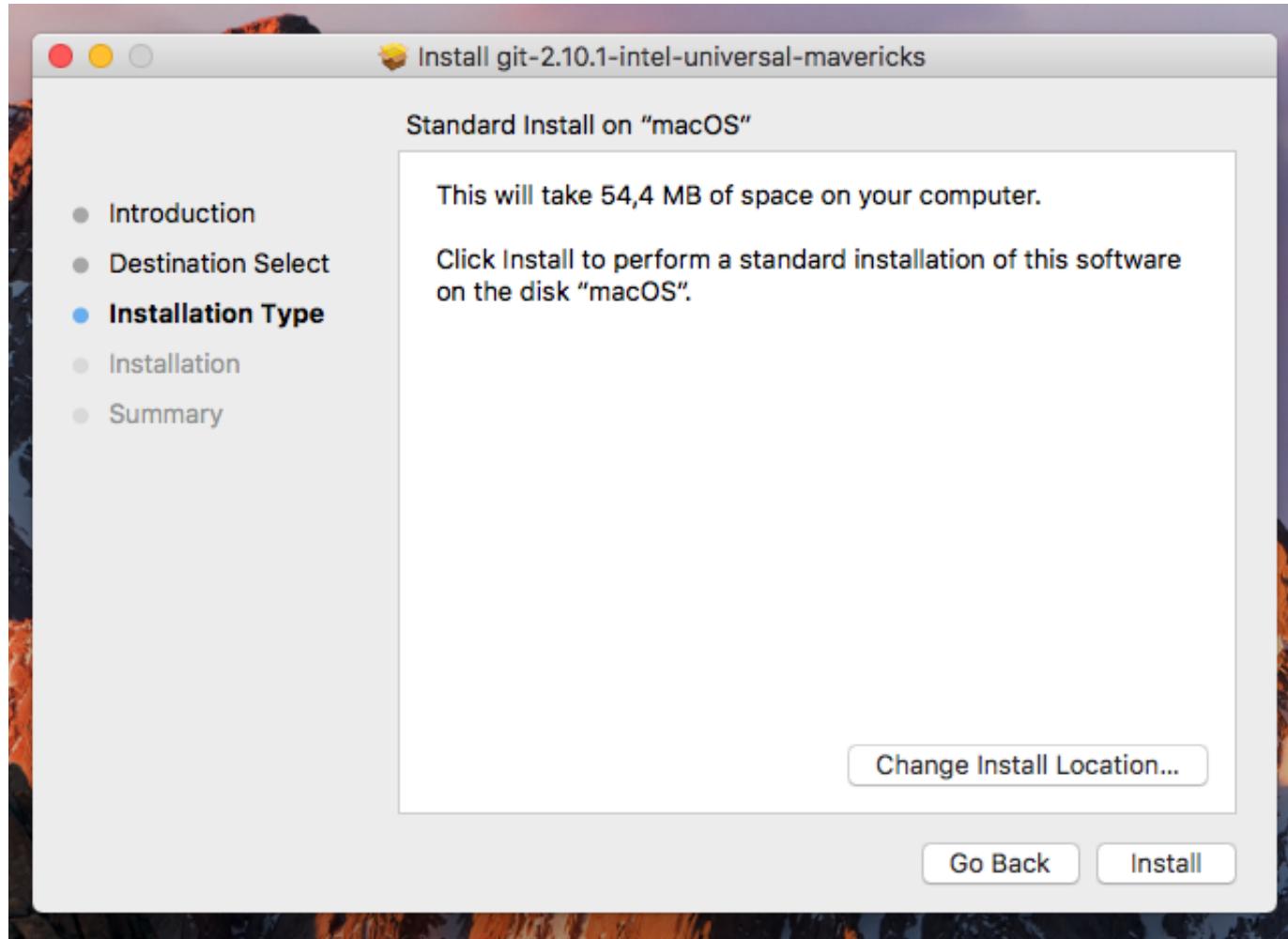
Hold down CTRL and click to let macOS prompt you to open the package

Installing Git on macOS



Let's start the
installation process

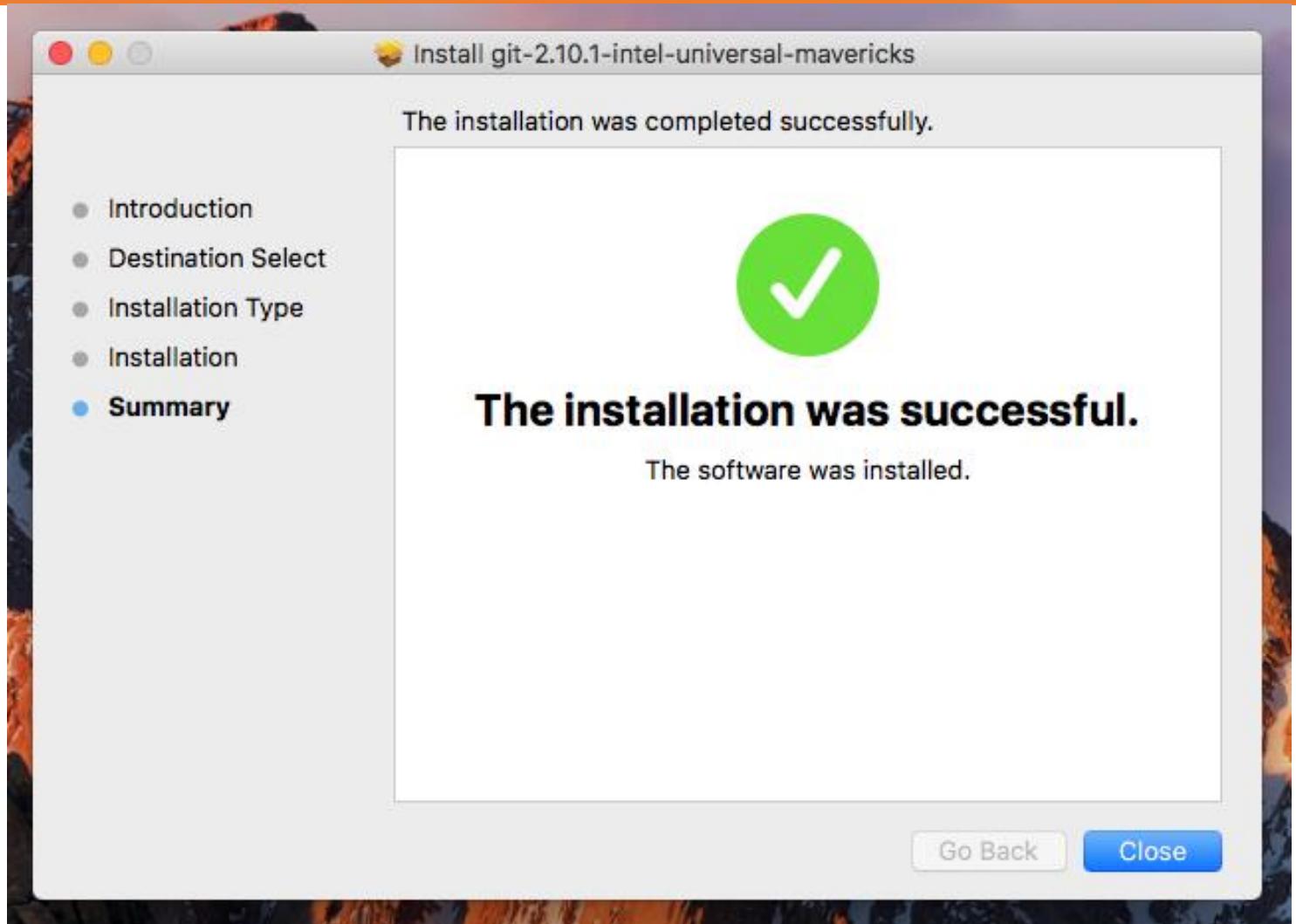
Installing Git on macOS



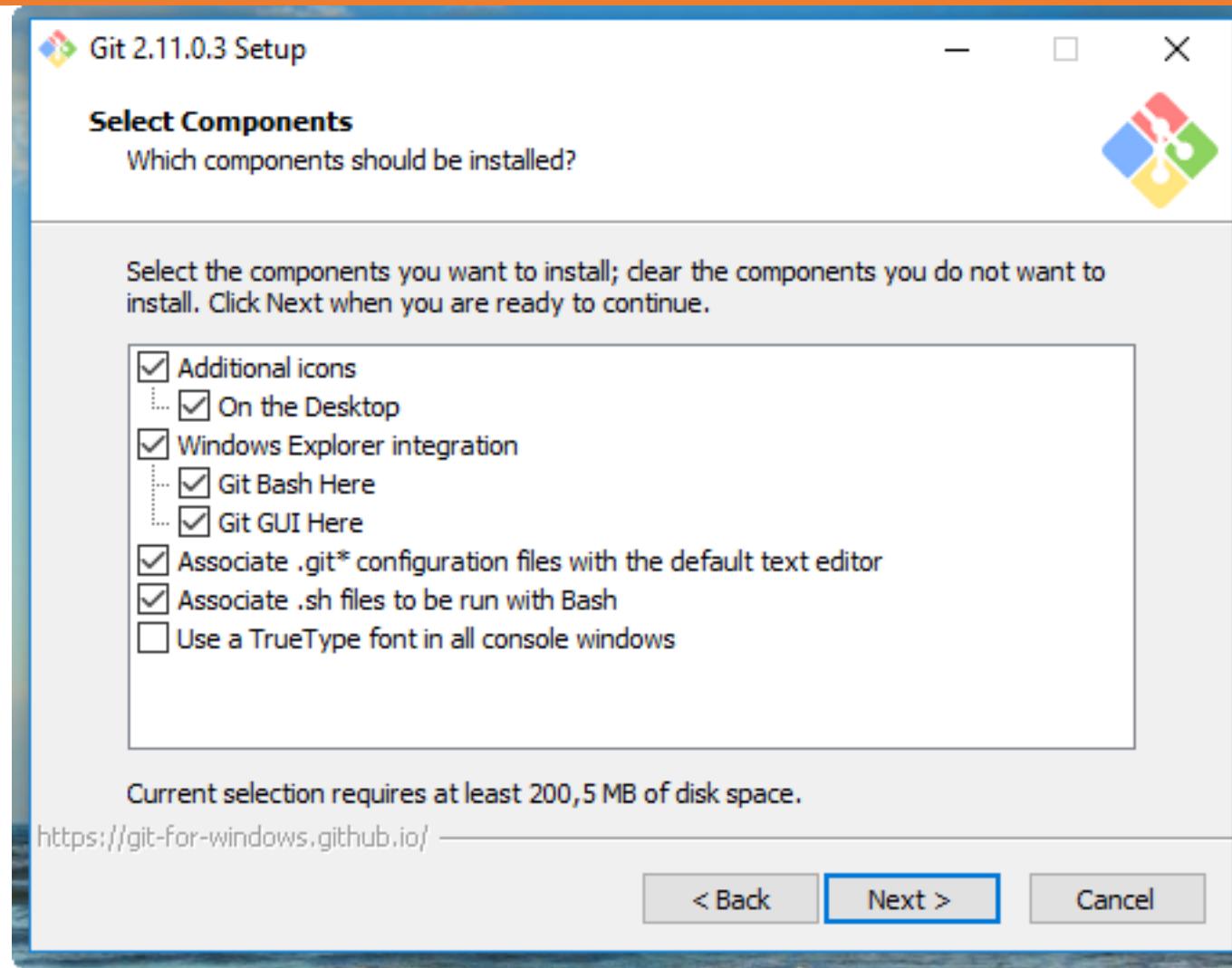
Here you can change the installation location, if you need; if in doubt, simply click Install

Installing Git on macOS

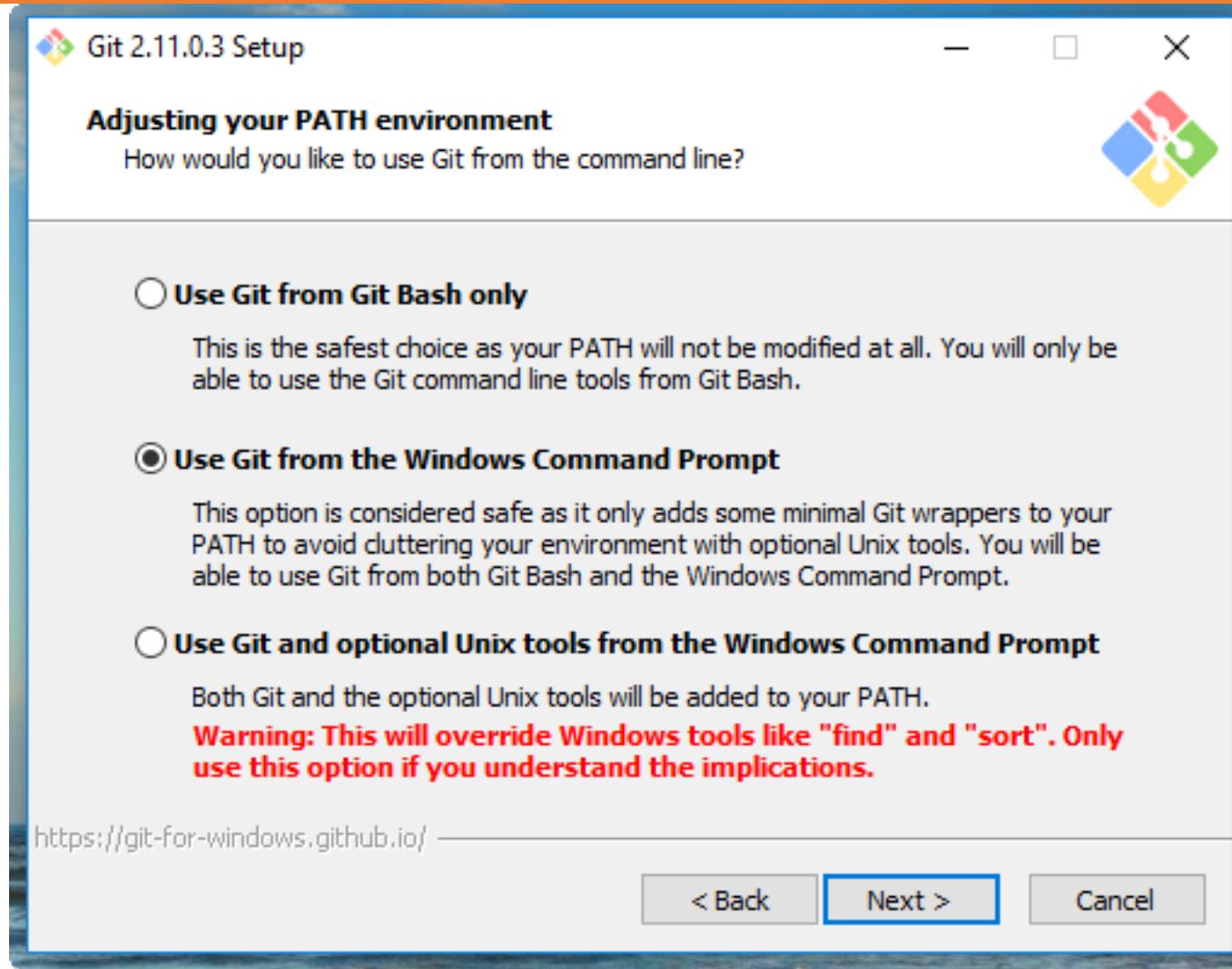
Installation complete



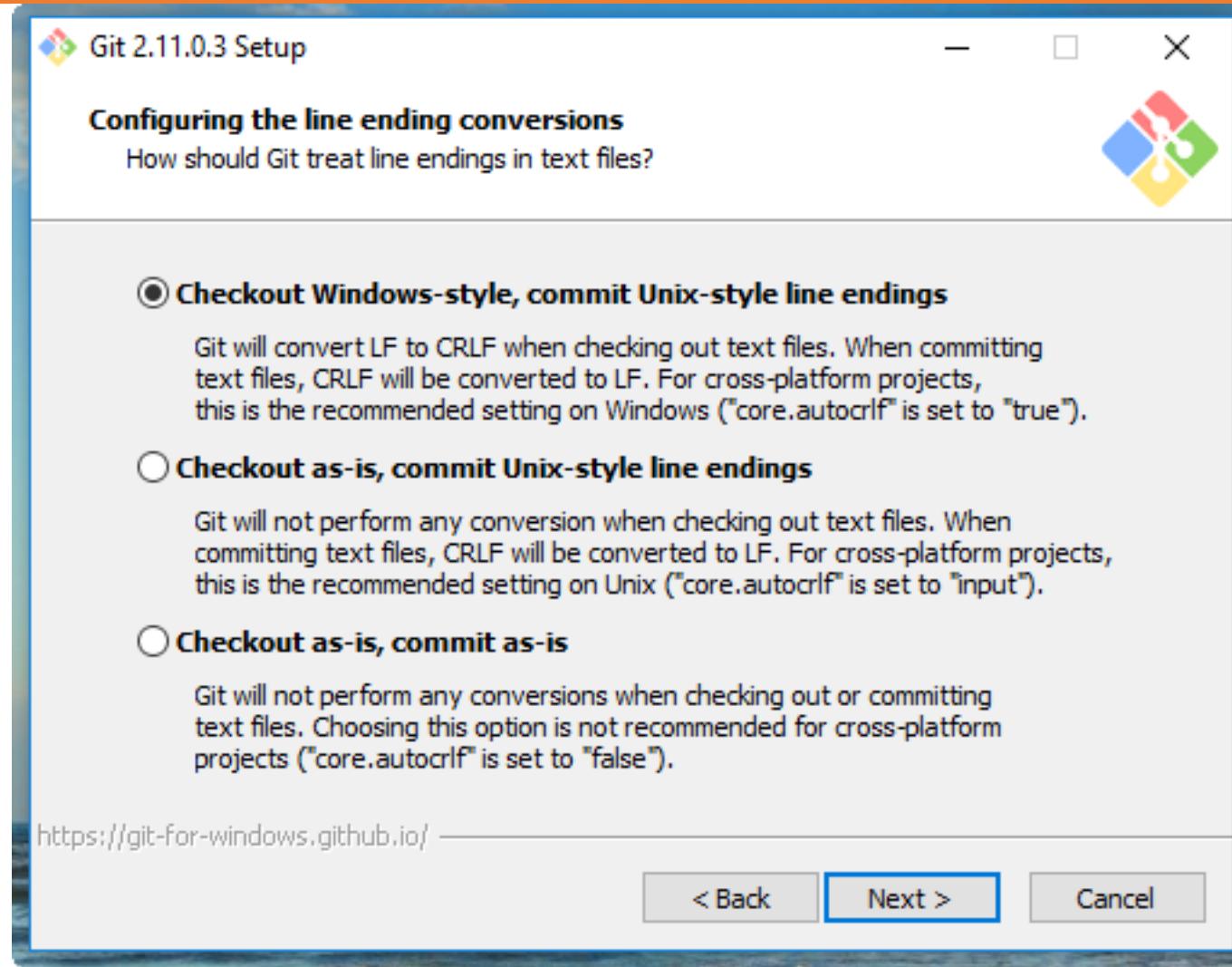
Installing Git on Windows



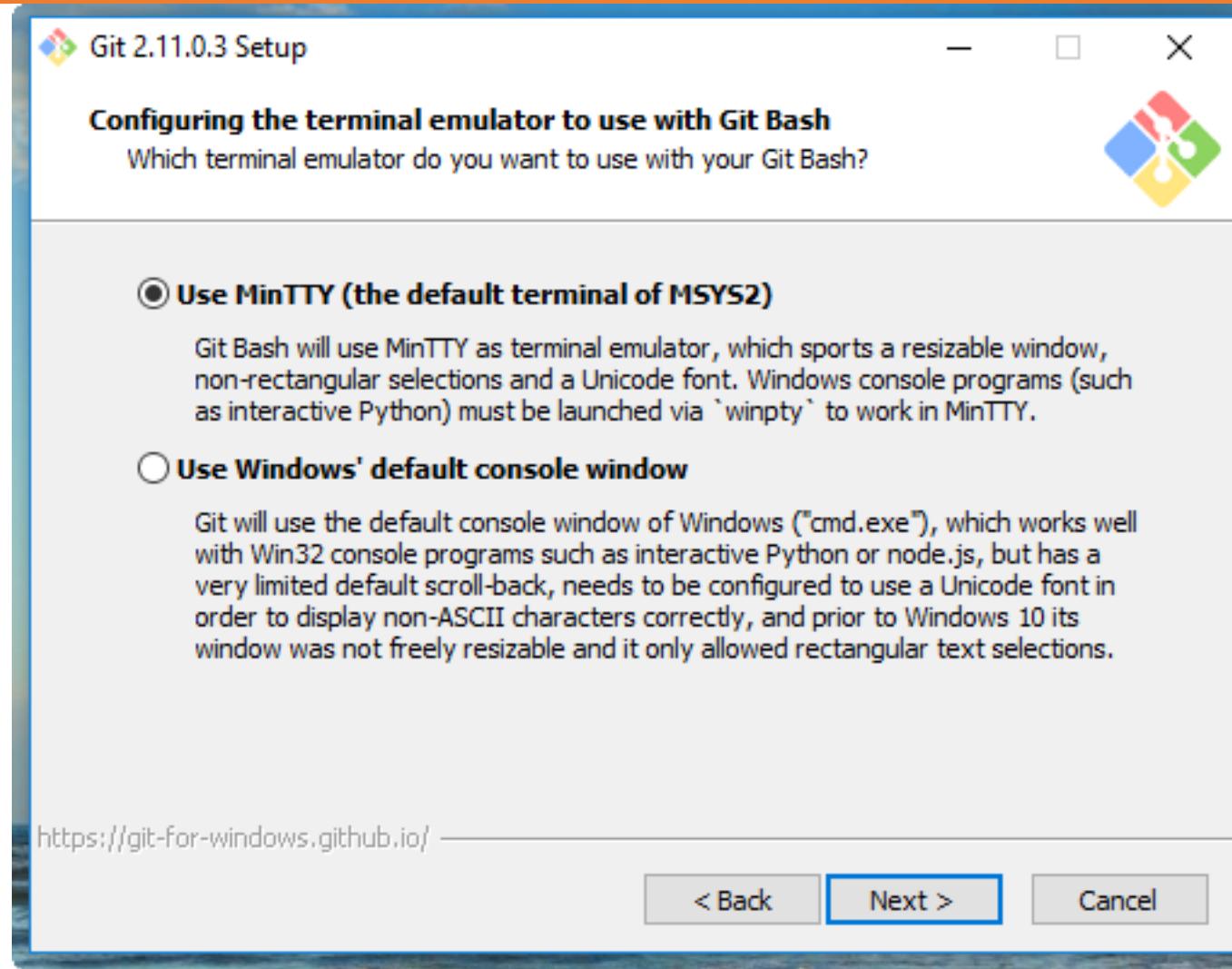
Installing Git on Windows



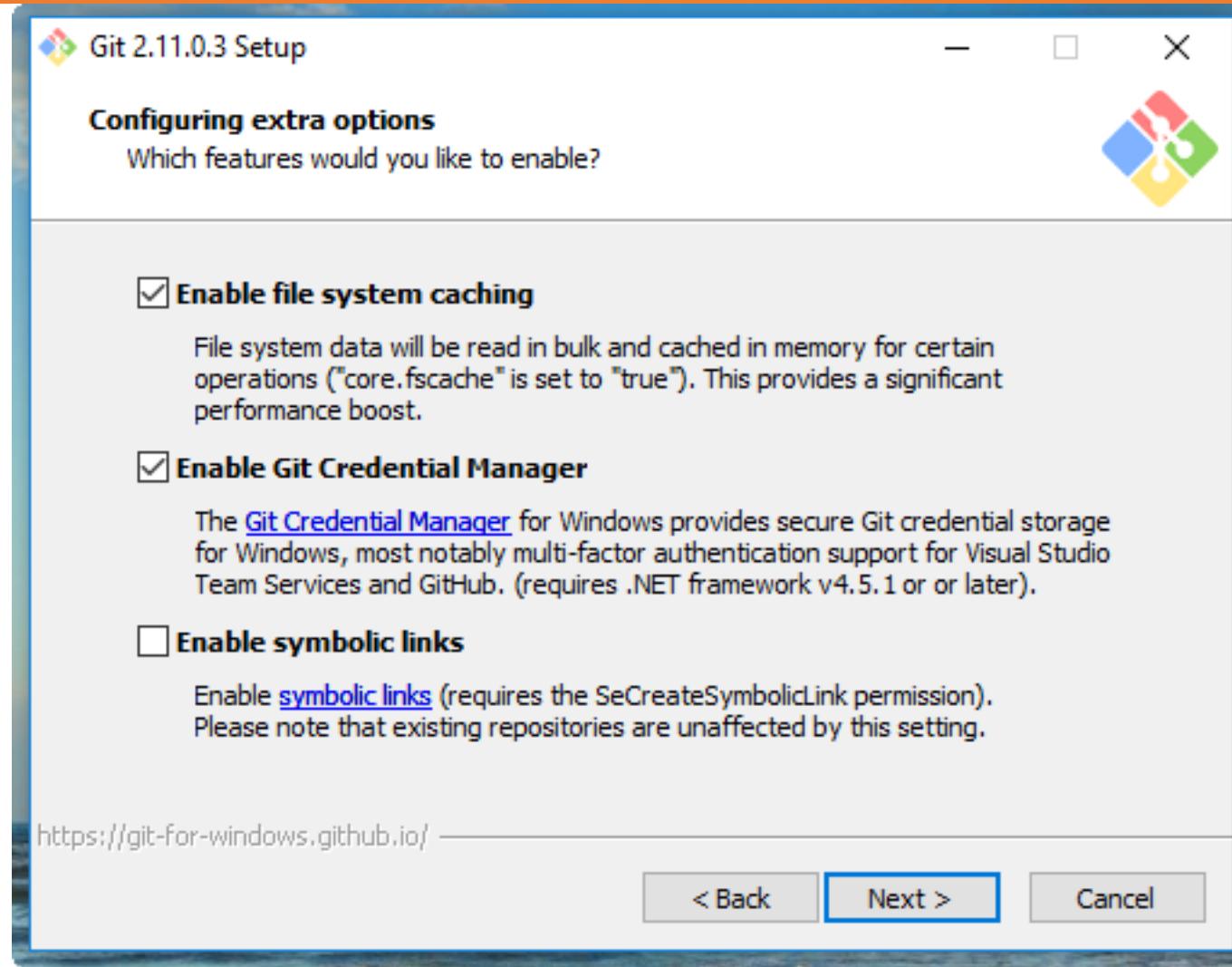
Installing Git on Windows



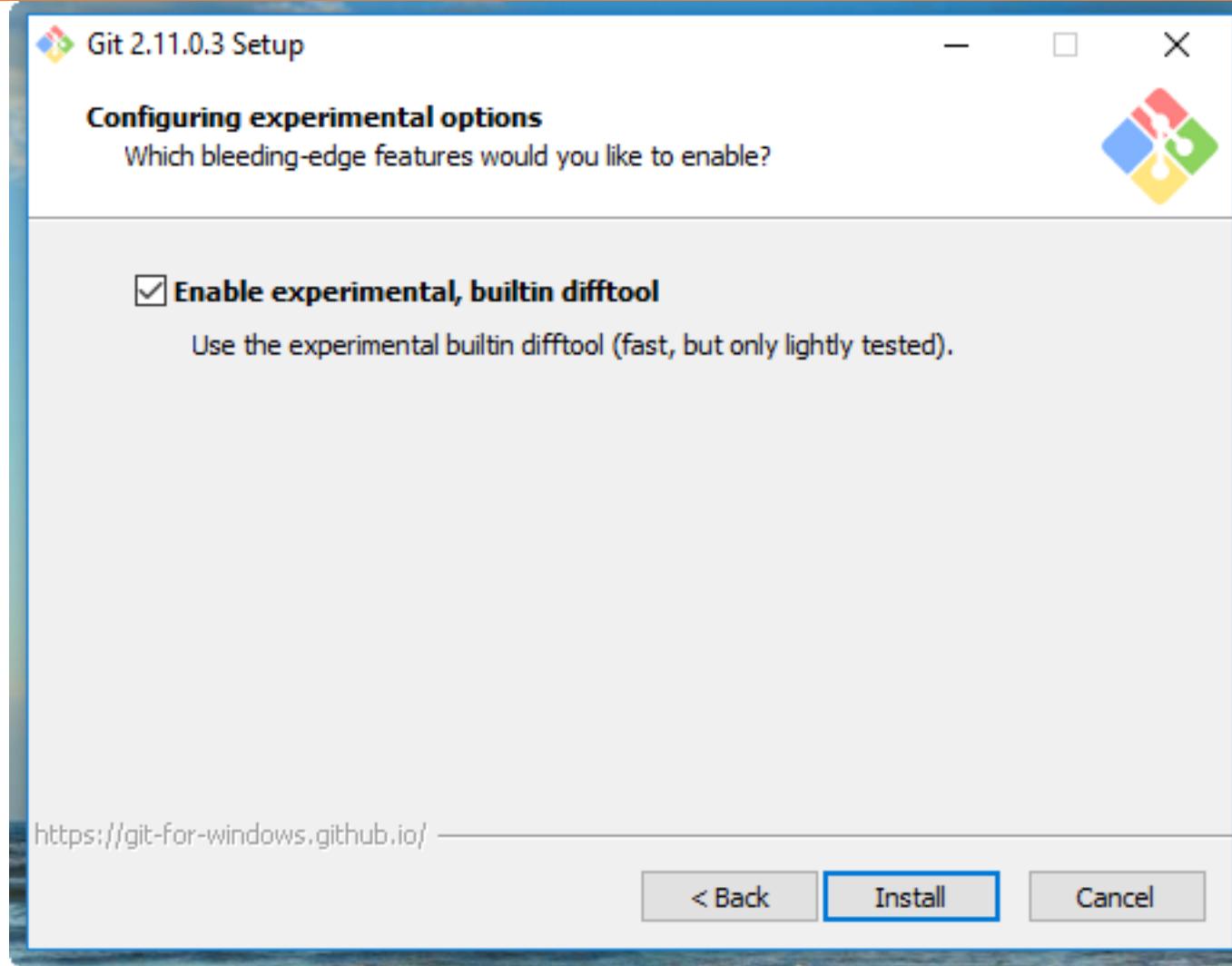
Installing Git on Windows



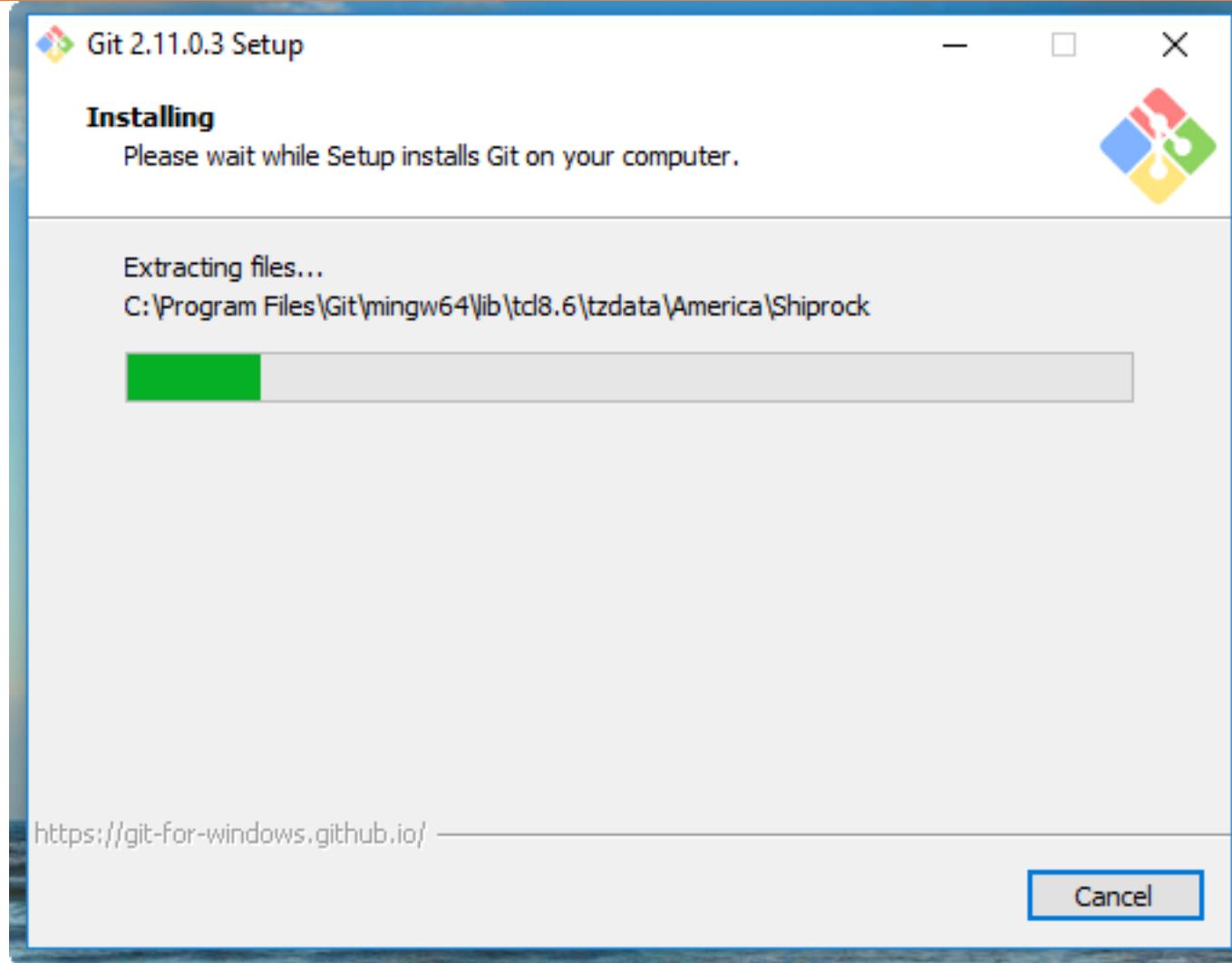
Installing Git on Windows



Installing Git on Windows

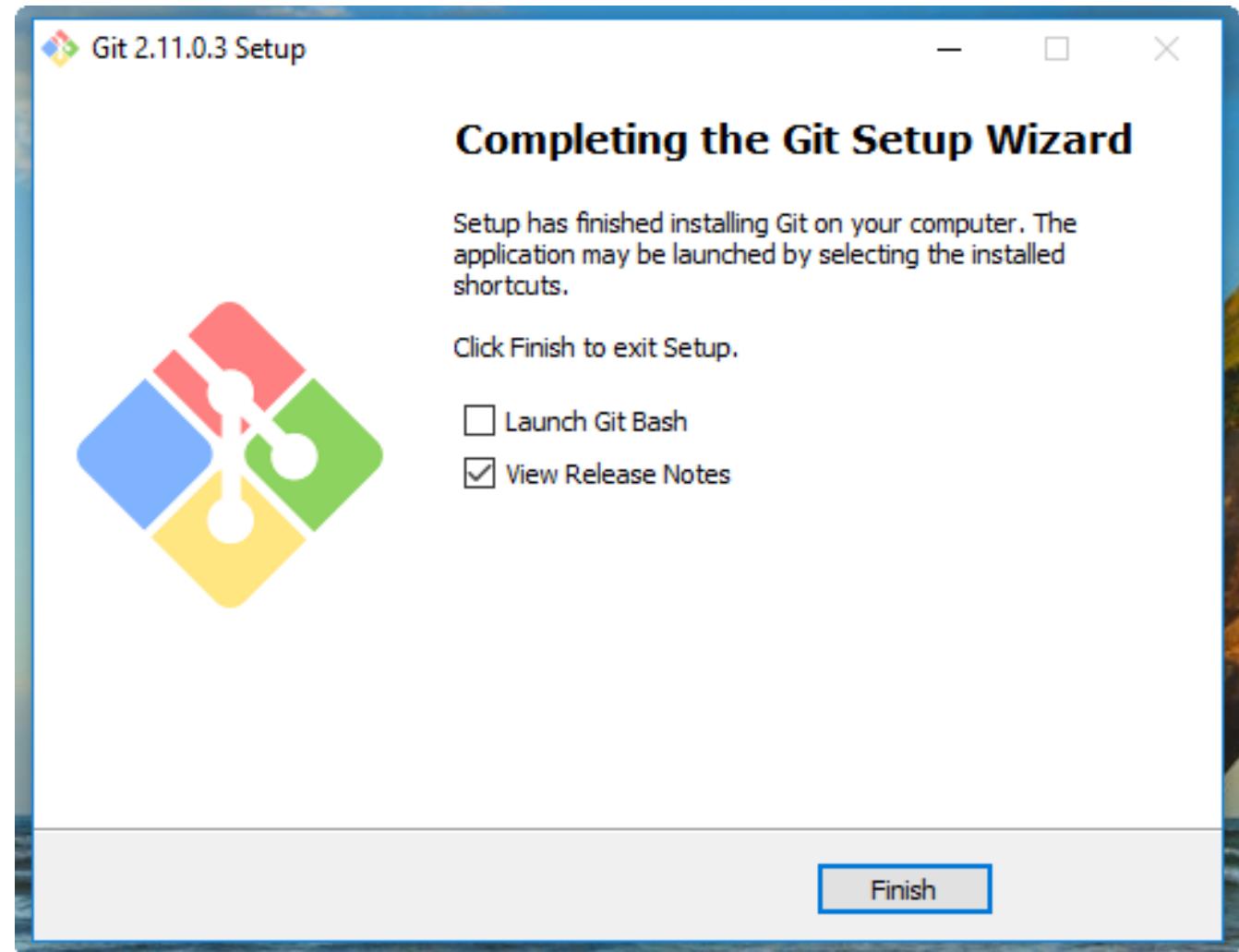


Installing Git on Windows



Installing Git on Windows

Git for Windows will install it in the default Program Files folder, as all the Windows programs usually do.

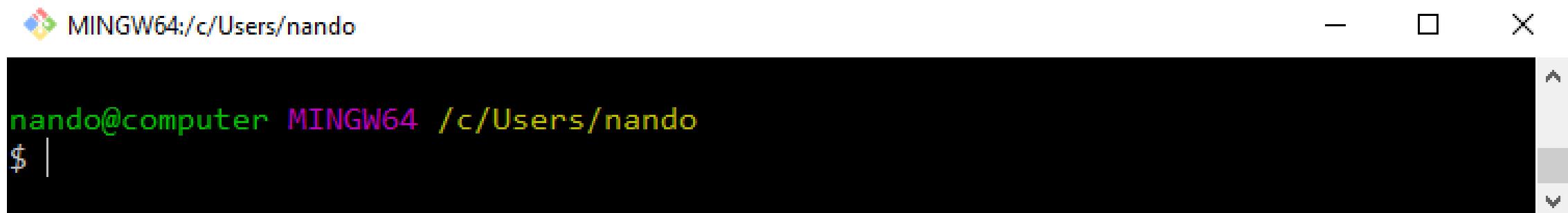


Running our first Git command

- Regardless of the OS in our screenshots, the git commands will work on Mac, Windows, Linux, etc.
- It's time to test our installation.
- Is Git ready to rock?
- ***Let's find out!***

Running our first Git command

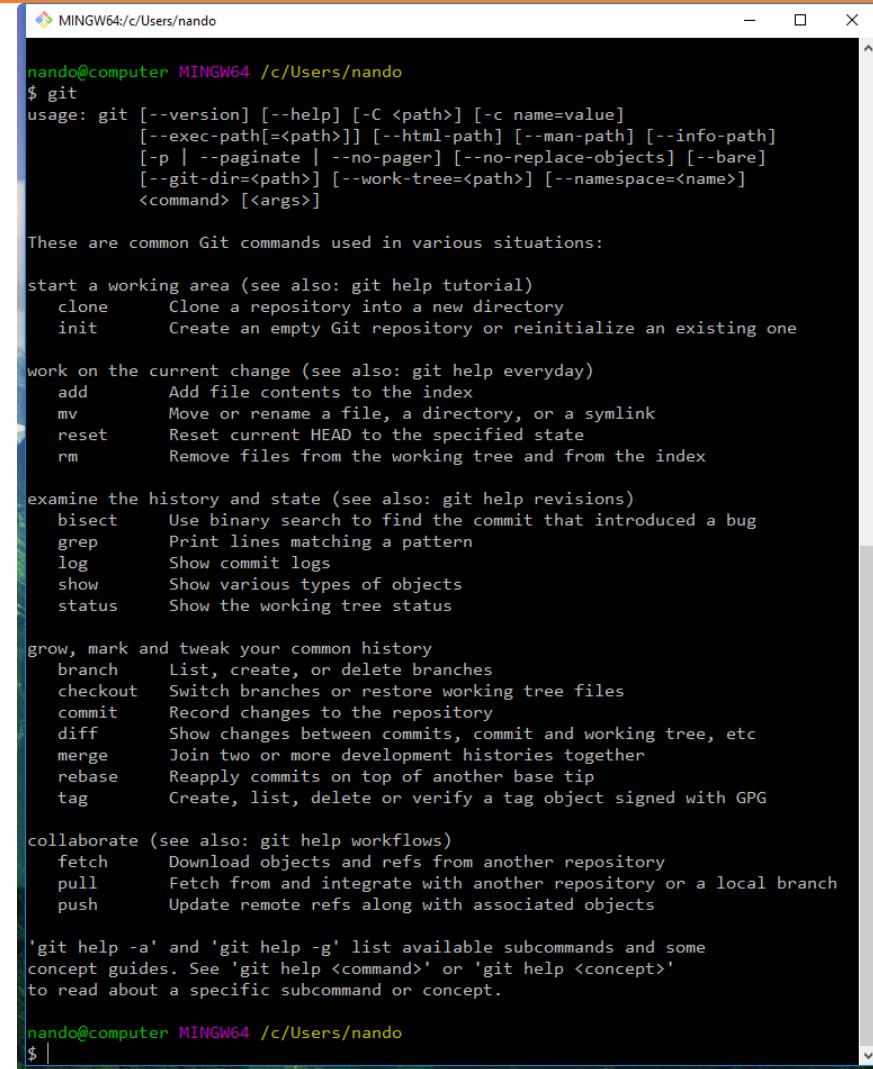
- Using shell integration, right-click on an empty place on the desktop and choose the new menu item Git Bash Here.
- It will appear as a new MinTTY shell, providing you a Git-ready bash for Windows:



A screenshot of a MinTTY terminal window titled "MINGW64:/c/Users/nando". The window shows a command-line interface with a green prompt: "nando@computer MINGW64 /c/Users/nando". Below the prompt is a blue dollar sign (\$) indicating where input can be entered. The window has standard operating system window controls (minimize, maximize, close) at the top right.

Running our first Git command

- Now that we have a new, shiny Bash prompt, simply type **git** (or the equivalent, **git --help**)



```
nando@computer MINGW64 /c/Users/nando
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
  add        Add file contents to the index
  mv         Move or rename a file, a directory, or a symlink
  reset     Reset current HEAD to the specified state
  rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
  bisect    Use binary search to find the commit that introduced a bug
  grep      Print lines matching a pattern
  log       Show commit logs
  show      Show various types of objects
  status    Show the working tree status

grow, mark and tweak your common history
  branch   List, create, or delete branches
  checkout Switch branches or restore working tree files
  commit   Record changes to the repository
  diff     Show changes between commits, commit and working tree, etc
  merge   Join two or more development histories together
  rebase   Reapply commits on top of another base tip
  tag     Create, list, delete or verify a tag object signed with GPG

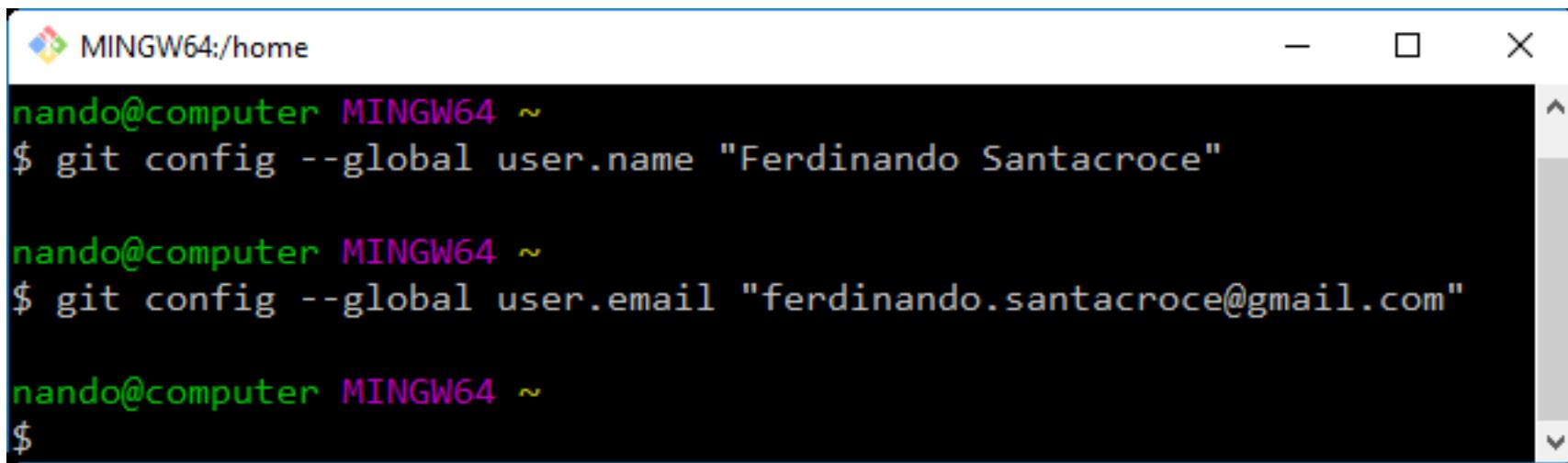
collaborate (see also: git help workflows)
  fetch   Download objects and refs from another repository
  pull    Fetch from and integrate with another repository or a local branch
  push    Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

nando@computer MINGW64 /c/Users/nando
$ |
```

Making presentations

- Git needs to know who you are this is because in Git, every modification you make in a repository has to be signed with the name and email of the author so, before doing anything else, we have to tell Git this information.
- Type these two commands:



The image shows a terminal window titled "MINGW64:/home". It contains the following text:

```
nando@computer MINGW64 ~
$ git config --global user.name "Ferdinando Santacroce"

nando@computer MINGW64 ~
$ git config --global user.email "ferdinando.santacroce@gmail.com"

nando@computer MINGW64 ~
$
```

Setting up a new repository

- The first step is to set up a new repository.
- A repository is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner.
- Physically, a repository is nothing other than a folder that contains a special .git folder, the folder where the magic happens.

Setting up a new repository



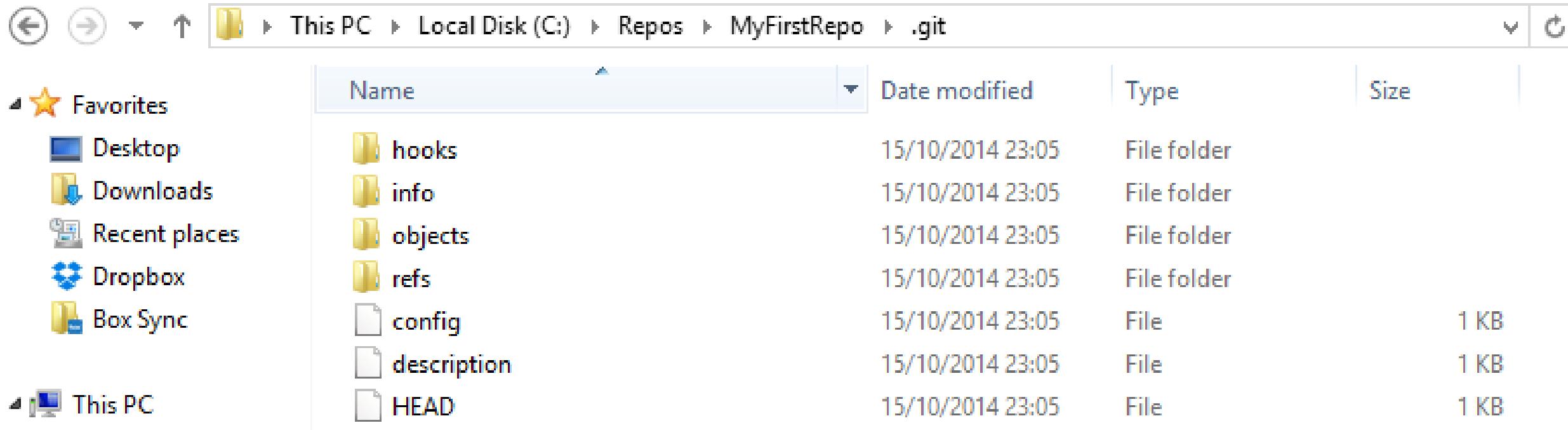
MINGW64:/c/Repos/MyFirstRepo

```
[1] ~
$ cd /c/Repos/MyFirstRepo/

[2] /c/Repos/MyFirstRepo
$ git init
Initialized empty Git repository in C:/Repos/MyFirstRepo/.git/

[3] /c/Repos/MyFirstRepo (master)
$ |
```

Setting up a new repository



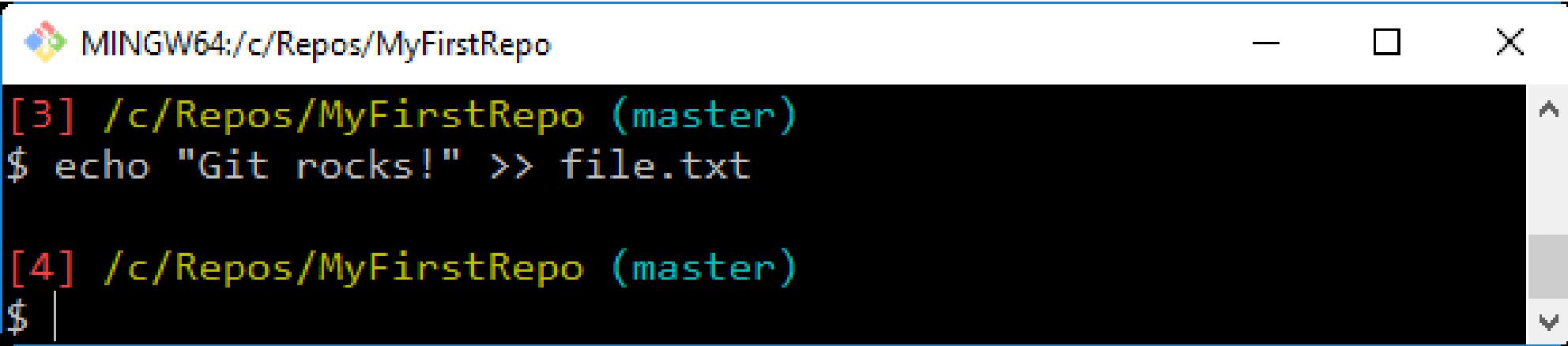
The screenshot shows a Windows File Explorer window with the following details:

- Address Bar:** This PC > Local Disk (C:) > Repos > MyFirstRepo > .git
- Favorites:** Desktop, Downloads, Recent places, Dropbox, Box Sync, This PC.
- File List:** The contents of the .git folder are displayed in a table format.

Name	Date modified	Type	Size
hooks	15/10/2014 23:05	File folder	
info	15/10/2014 23:05	File folder	
objects	15/10/2014 23:05	File folder	
refs	15/10/2014 23:05	File folder	
config	15/10/2014 23:05	File	1 KB
description	15/10/2014 23:05	File	1 KB
HEAD	15/10/2014 23:05	File	1 KB

Adding a file

- Let's create a text file, just to give it a try:



The screenshot shows a terminal window titled "MINGW64:/c/Repos/MyFirstRepo". The window contains the following command history:

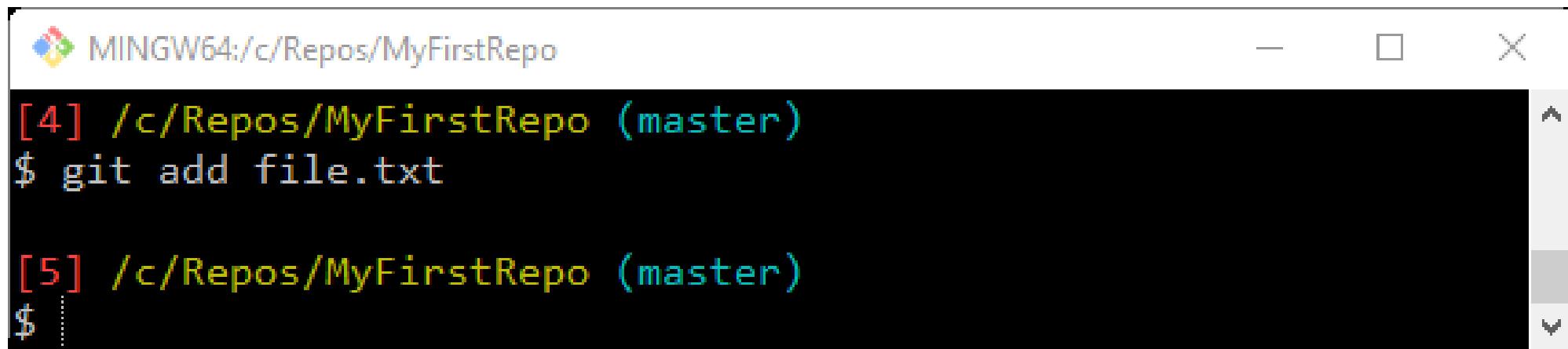
```
[3] /c/Repos/MyFirstRepo (master)
$ echo "Git rocks!" >> file.txt

[4] /c/Repos/MyFirstRepo (master)
$ |
```

The terminal window has a standard Windows-style title bar with minimize, maximize, and close buttons. The main area is a black terminal window with white text.

Adding a file

- Okay, back to the topic. I want file.txt under the control of Git, so let's add it, as shown here:



The screenshot shows a terminal window titled "MINGW64:/c/Repos/MyFirstRepo". The window contains the following text:

```
[4] /c/Repos/MyFirstRepo (master)
$ git add file.txt

[5] /c/Repos/MyFirstRepo (master)
$
```

Adding a file

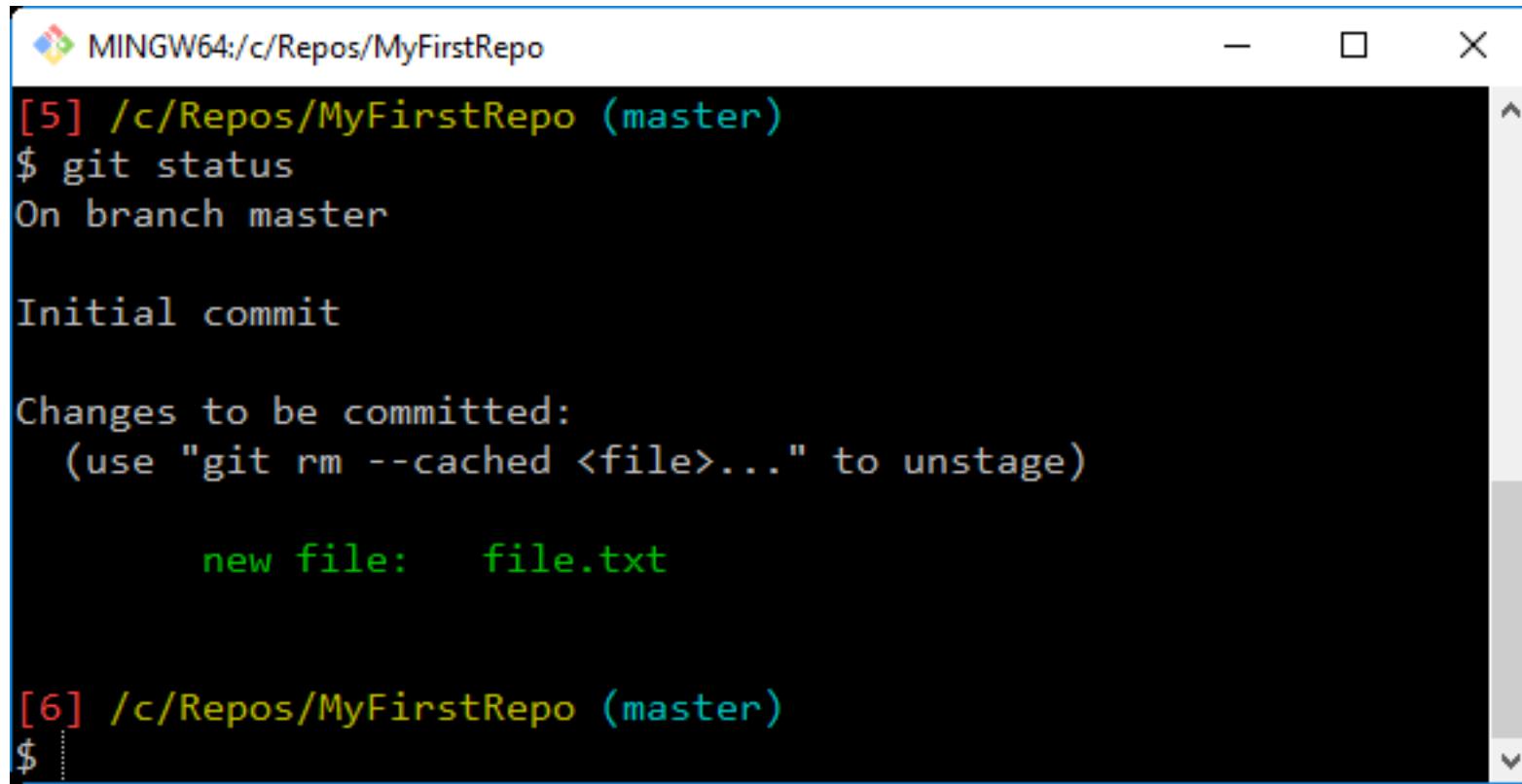
- In response to this command, it could happen that you will see this response message from Git:

warning: LF will be replaced by CRLF in file.txt.

- The file will have its original line endings in your working directory.

Adding a file

- Using the git status command, we can check the status of the repository, as shown in this screenshot:



```
MINGW64:/c/Repos/MyFirstRepo
[5] /c/Repos/MyFirstRepo (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

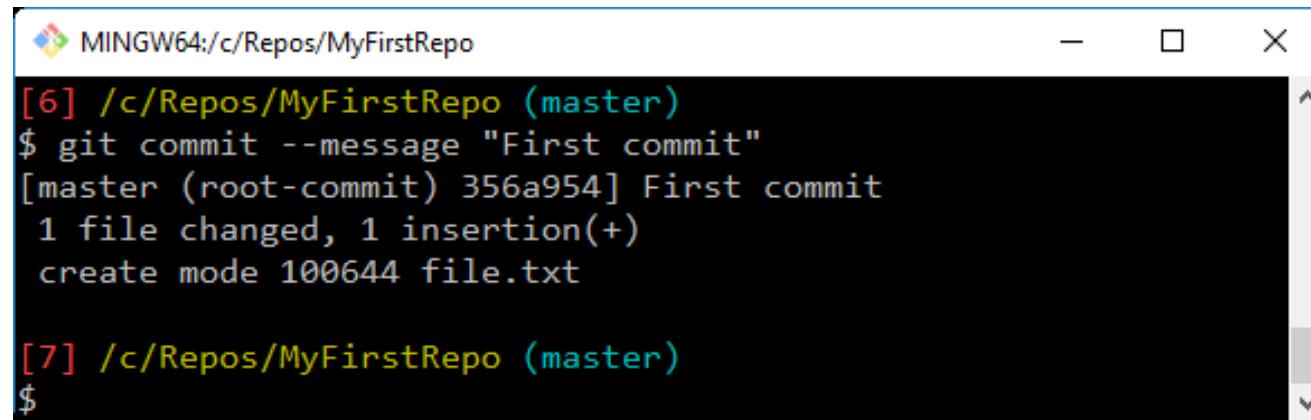
    new file:   file.txt

[6] /c/Repos/MyFirstRepo (master)
$
```

A screenshot of a terminal window titled 'MINGW64:/c/Repos/MyFirstRepo'. The window shows the output of the 'git status' command. It indicates that the user is on the 'master' branch and performing an 'Initial commit'. A single file named 'file.txt' is listed as a 'new file' to be committed. The terminal window has a standard title bar with minimize, maximize, and close buttons.

Committing the added file

- At this point, Git knows about file.txt, but we have to perform another step to fix the snapshot of its content.
- We have to commit it using the appropriate git commit command.
- This time, we will add some flavor to our command, using the --message (or -m) subcommand, as shown here:



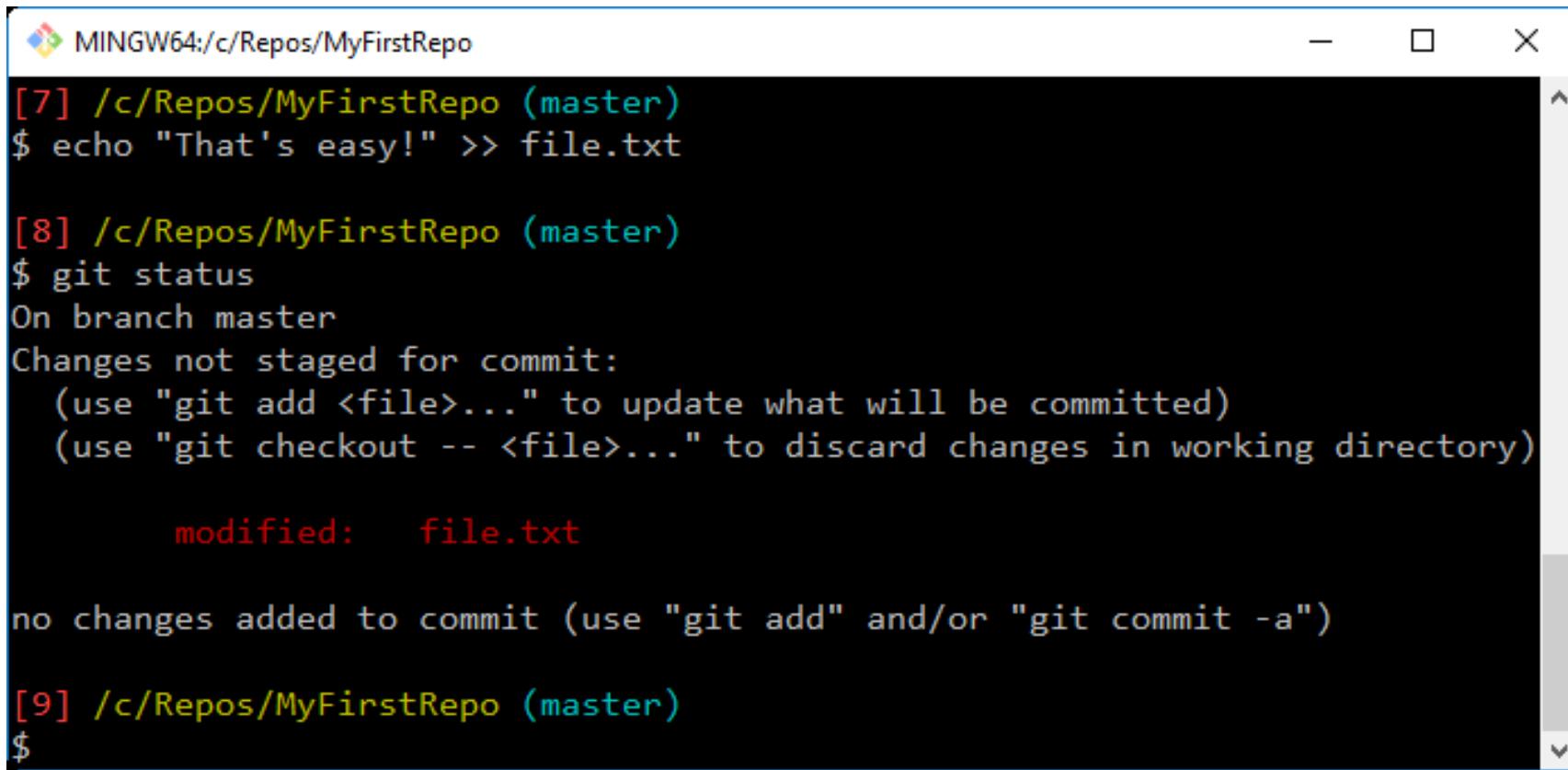
The screenshot shows a terminal window titled 'MINGW64:/c/Repos/MyFirstRepo'. The command entered is '\$ git commit --message "First commit"'. The output shows a successful commit: '[master (root-commit) 356a954] First commit' with '1 file changed, 1 insertion(+)' and 'create mode 100644 file.txt'. The window has a standard title bar with minimize, maximize, and close buttons.

```
[6] /c/Repos/MyFirstRepo (master)
$ git commit --message "First commit"
[master (root-commit) 356a954] First commit
 1 file changed, 1 insertion(+)
   create mode 100644 file.txt

[7] /c/Repos/MyFirstRepo (master)
$
```

Modifying a committed file

- Now, we can try to make some modifications to the file and see how to deal with it, as shown in the following screenshot:



The screenshot shows a terminal window titled "MINGW64:/c/Repos/MyFirstRepo". The terminal history is as follows:

- [7] /c/Repos/MyFirstRepo (master)
\$ echo "That's easy!" >> file.txt
- [8] /c/Repos/MyFirstRepo (master)
\$ git status
On branch master
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: file.txt
- [9] /c/Repos/MyFirstRepo (master)
\$

The terminal shows that a new line was added to the file "file.txt". When "git status" was run, it indicated that the file had been modified but was not staged for commit. The terminal prompt is at the end of line 9.

Modifying a committed file

- So, let's add the file again for the purpose of getting things ready for the next commit:

```
MINGW64:/c/Repos/MyFirstRepo
[9] /c/Repos/MyFirstRepo (master)
$ git commit
On branch master
Changes not staged for commit:
  modified:   file.txt

no changes added to commit

[10] /c/Repos/MyFirstRepo (master)
$ git add file.txt

[11] /c/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   file.txt

[12] /c/Repos/MyFirstRepo (master)
$ |
```

Summary

- In this lesson, you learned that Git is not so difficult to install, even on a non-Unix platform, such as Windows.
- Once you have chosen a directory to include in a Git repository, you can see that initializing a new Git repository is as easy as executing a `git init` command.

2. Git Fundamentals Working Locally



Git Fundamentals - Working Locally

- In this lesson, we will dive deep into some of the fundamentals of Git; it is essential to understand well how Git thinks about files, its way of tracking the history of commits, and all the basic commands that we need to master, in order to become proficient.

Digging into Git internals

- I want to show you how Git works internally with only the help of the shell, allowing you to follow all the steps on your computer and hoping that these will be clear enough for you to understand.

Git objects

- Let's create a new repository to refresh our memory and then start learning a little bit more about Git.
- In this example, we use Git to track our shopping list before going to the grocery; so, create a new grocery folder, and then initialize a new Git repository:

```
[1] ~$ mkdir grocery
```

```
[2] ~$ cd grocery/
```

```
[3] ~/grocery$ git init
```

```
Initialized empty Git repository in C:/Users/san/Google  
Drive/LV/PortableGit/home/grocery/.git/
```

Git objects

- As we have already seen before, the result of the git init command is the creation of a .git folder, where Git stores all the files it needs to manage our repository:

```
[4] ~/grocery (master)$ ls -althr
total 8drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 ./drwxr-
xr-x 1 san 1049089 0 Aug 17 11:11 ../drwxr-xr-x 1 san
1049089 0 Aug 17 11:11 .git/
```

Git objects

- Go on and create a new README.md file to remember the purpose of this repository:

```
[5] ~/grocery (master)$ echo "My shopping list repository"  
> README.md
```

- Then add a banana to the shopping list:

```
[6] ~/grocery (master)$ echo "banana" > shoppingList.txt
```

Git objects

- At this point, as you already know, before doing a commit, we have to add files to the staging area; add both the files using the shortcut git add :

[7] ~/grocery (master)\$ git add .

- With this trick (the dot after the git add command), you can add all the new or modified files in one shot.
- At this point, if you didn't set up a global username and email like we did in lesson1, Getting Started with Git, this is a thing that could happen:

Refer to the file 2_1.txt: <https://jmp.sh/2g5H62t>

Git objects

- So, let's change these settings and amend our commit (amending a commit is a way to redo the last commit and fix up some little mistakes, such as adding a forgotten file, changing the message or the author, as we are going to do; later we will learn in detail what this means):

```
[9] ~/grocery (master)$ git config user.name "Ernesto Lee"
```

```
[10] ~/grocery (master)$ git config user.email socrates73@gmail.com
```

Git objects

- For the purpose of this exercise, please leave the message as it is, press Esc, and then input the :wq (or :x) command and press Enter to save and exit:

```
[11] ~/grocery (master)$ git commit --amend --reset-author  
#here Vim opens[master a57d783] Add a banana to the shopping list 2 files changed, 2 insertions(+) create mode 100644 README.md create mode 100644 shoppingList.txt
```

Git objects

- Now it's time to start investigating commits.
- To verify the commit we have just created, we can use the git log command:

```
[12] ~/grocery (master)$ git log  
commit a57d783905e6a35032d9b0583f052fb42d5a1308  
Author: Ernesto Lee socrates73@gmail.com  
Date: Thu Aug 17 13:51:33 2017 +0200Add a banana to the shopping  
list
```

Git objects

- Just under the author and date, after a blank line, we can see the message we attached to the commit we made; even the message is part of the commit itself but there's something more under the hood; let's try to use the git log command with the --format=fuller option:

```
[13] ~/grocery (master)$ git log --format=fuller
commit a57d783905e6a35032d9b0583f052fb42d5a1308
Author: Ernesto Lee <socrates73@gmail.com>AuthorDate: Thu Aug 17
13:51:33 2017 +0200Commit: Ernesto Lee socrates73@gmail.com
CommitDate: Thu Aug 17 13:51:33 2017 +0200Add a banana to the
shopping list
```

Git objects

- We analyzed a commit, and the information supplied by a simple git log; but we are not yet satisfied, so go deeper and see what's inside.
- Using the git log command again, we can enable x-ray vision using the --format=raw option:

```
[14] ~/grocery (master)$ git log --format=raw
commit a57d783905e6a35032d9b0583f052fb42d5a1308tree
a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ernesto Lee <socrates73@gmail.com> 1502970693
+0200committer Ernesto Lee <socrates73@gmail.com> 1502970693
+0200Add a banana to the shopping list
```

Git objects

- Back on topic; type the command, specifying the first characters of the commit's hash (a57d7 in my case):

```
[15] ~/grocery (master)$ git cat-file -p a57d7  
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40author  
Ernesto Lee <socrates73@gmail.com> 1502970693  
+0200committer Ernesto Lee <socrates73@gmail.com>  
1502970693 +0200Add a banana to the shopping list
```

Git objects

Porcelain commands and plumbing commands

- Git, as we know, has a myriad of commands, some of which are practically never used by the average user; as by example, the previous git cat-file.
- These commands are called plumbing commands, while those we have already learned about, such as git add, git commit, and so on, are among the so-called porcelain commands.

Git objects

- Here, for convenience, there is the output of the git cat-file -p command typed before:

```
[15] ~/grocery (master)$ git cat-file -p a57d7  
tree  
a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40author  
Ernesto Lee <socrates73@gmail.com> 1502970693  
+0200committer Ernesto Lee <socrates73@gmail.com>  
1502970693 +0200
```

Trees

- The tree is a container for blobs and other trees.
- The easiest way to understand how it works is to think about folders in your operating system, which also collect files and other subfolders inside them.
- Let's try to see what this additional Git object holds, using again the git cat-file -p command:
 - [16] ~/grocery (master)\$ git cat-file -p a31c3100644
blob907b75b54b7c70713a79cc6b7b172fb131d3027d
README.md100644 blob
637a09b86af61897fb72f26fb874f2ae726db82 shoppingList.txt

Blobs

```
[17] ~/grocery (master)$ git cat-file -p 637a0  
banana
```

- Wow! Its content is exactly the content of our shoppingFile.txt file.
- To confirm, we can use the cat command, which on *nix systems allows you to see the contents of a file:

```
[18] ~/grocery (master)$ cat shoppingList.txt  
banana
```

- As you can see, the result is the same.

Blobs

- Blobs are binary files, nothing more and nothing less.
- These byte sequences, which cannot be interpreted with the naked eye, retain inside information belonging to any file, whether binary or textual, images, source code, archives, and so on.
- Everything is compressed and transformed into a blob before archiving it into a Git repository.

Blobs

- Let's try to understand it better with an example.
- Open a shell and try to play a bit with another plumbing command, git hash-object:

```
[19] ~/grocery (master)$ echo "banana" | git hash-object --stdin  
637a09b86af61897fb72f26fb874f2ae726db82
```

Even deeper - the Git storage object model

- Do you remember the .git folder? Let's put our nose inside it:

```
[20] ~/grocery (master)$ ls -althr
```

Even deeper - the Git storage object model

- Within it, there is an objects subfolder; let's take a look:

```
[21] ~/grocery (master) $ ll  
.git/objects/ total 4drwxr-xr-x 1 san 1049089 0 Aug 18 17:15  
./drwxr-xr-x 1 san 1049089 0 Aug 18 17:22 ../drwxr-xr-x 1 san  
1049089 0 Aug 18 17:15 63/drwxr-xr-x 1 san 1049089 0 Aug 18  
17:15 90/drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 a3/drwxr-xr-x 1  
san 1049089 0 Aug 18 17:15 a5/drwxr-xr-x 1 san 1049089 0 Aug  
18 17:15 c7/drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 info/drwxr-  
xr-x 1 san 1049089 0 Aug 18 17:12 pack/
```

Even deeper - the Git storage object model

- Other than info and pack folders, which are not interesting for us right now, as you can see there are some other folders with a strange two-character name; let's go inside the 63 folder:

```
[22] ~/grocery (master)
$ ll .git/objects/63/
total 1
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 .
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ..
-r--r--r-- 1 san 1049089 20 Aug 17 13:34
7a09b86af61897fb72f26fb874f2ae726db82
```

Even deeper - the Git storage object model

- To become aware of this, we need a new commit. So, let's now proceed modifying the shoppingList.txt file:

```
[23] ~/grocery (master)$ echo "apple" >>  
shoppingList.txt
```

```
[24] ~/grocery (master)$ git add shoppingList.txt
```

```
[25] ~/grocery (master)$ git commit -m "Add an apple"
```

```
[master e4a5e7b] Add an apple 1 file changed, 1  
insertion(+)
```

Even deeper - the Git storage object model

- Use the git log command to check the new commit; the --oneline option allows us to see the log in a more compact way:

```
[26] ~/grocery (master)$ git log --oneline  
e4a5e7b Add an applea57d783 Add a banana to the  
shopping list
```

Even deeper - the Git storage object model

- Okay, we have a new commit, with its hash.
- Time to see what's inside it:

```
[27] ~/grocery (master)$ git cat-file -p e4a5e7b  
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0parent  
a57d783905e6a35032d9b0583f052fb42d5a1308author  
Ernesto Lee <socrates73@gmail.com> 1503586854  
+0200committer Ernesto Lee <socrates73@gmail.com>  
1503586854 +0200Add an apple
```

Git doesn't use deltas

- Instead, in Git even if you change only a char in a big text file, it always stores a new version of the file: Git doesn't do deltas (at least not in this case), and every commit is actually a snapshot of the entire repository.
- At this point, people usually exclaim:

"Gosh, Git waste a large amount of disk space in vain!"

- Well, this is simply untrue.

Git doesn't use deltas

- Furthermore, Git has a clever way to deal with files; let's take a look again at the last commit:

```
[28] ~/grocery (master)$ git cat-file -p e4a5e7b  
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0parent  
a57d783905e6a35032d9b0583f052fb42d5a1308author  
Ernesto Lee <socrates73@gmail.com> 1503586854  
+0200committer Ernesto Lee <socrates73@gmail.com>  
1503586854 +0200Add an apple
```

Git doesn't use deltas

- Okay, now to the tree:

```
[29] ~/grocery (master)$ git cat-file -p 4c931e9100644  
blob 907b75b54b7c70713a79cc6b7b172fb131d3027d  
README.md100644 blob  
e4ceb844d94edba245ba12246d3eb6d9d3aba504  
shoppingList.txt
```

Git doesn't use deltas

- Annotate the two hashes on a notepad; now we have to look at the tree of the first commit; cat-file the commit:

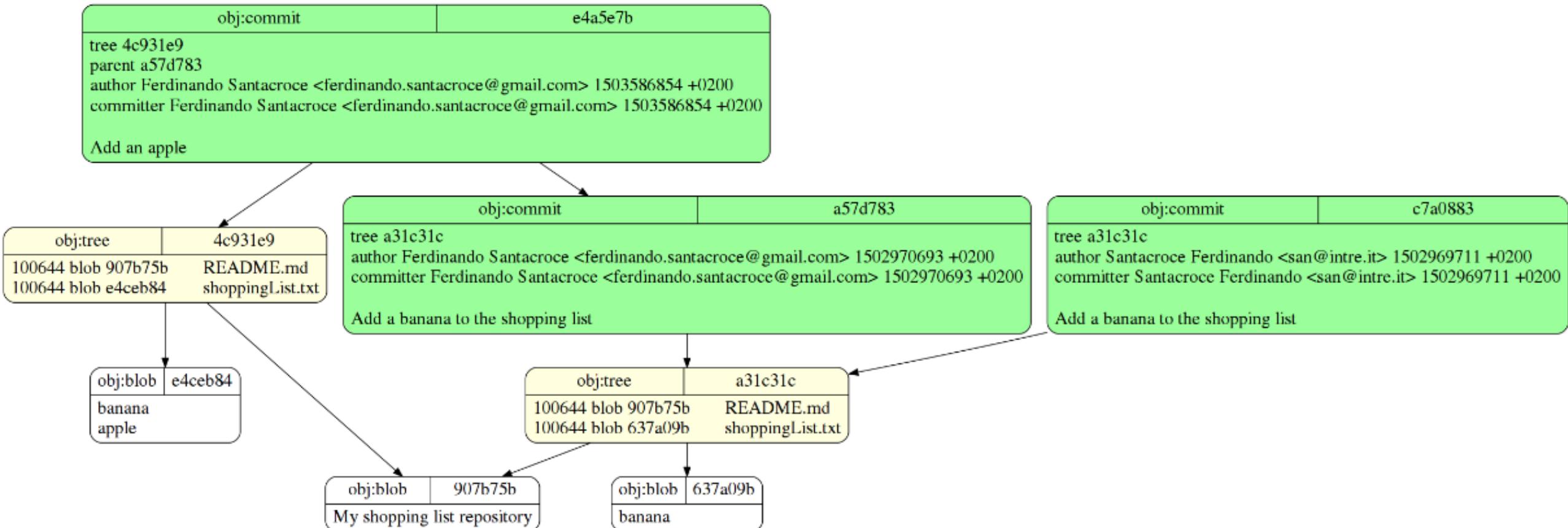
```
[30] ~/grocery (master)$ git cat-file -p a57d783  
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40author  
Ernesto Lee <socrates73@gmail.com> 1502970693  
+0200committer Ernesto Lee <socrates73@gmail.com>  
1502970693 +0200  
Add a banana to the shopping list
```

Git doesn't use deltas

- Then cat-file the tree:

```
[31] ~/grocery (master)$ git cat-file -p a31c31c100644  
blob 907b75b54b7c70713a79cc6b7b172fb131d3027d  
README.md100644 blob  
637a09b86af61897fb72f26fb874f2ae726db82  
shoppingList.txt
```

Wrapping up



Wrapping up

- In this graphic representation of previous slide, you will find a detailed diagram that represents the current structure of the newly created repository; you can see trees (yellow), blobs (white), commits (green), and all relationships between them, represented by oriented arrows.

It's all about labels

- In Git, a branch is nothing more than a label, a mobile label placed on a commit.
- In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

Git references

- Let's start exploring this topic by checking the current status of our grocery repository; we do it using the well-known git log command, this time adding some new options:

```
[1] ~/grocery (master)$ git log --oneline --graph --decorate  
e4a5e7b  
(HEAD -> master) Add an apple* a57d783 Add a banana  
to the shopping list
```

Git references

- We'll now do a new commit and see what happens:

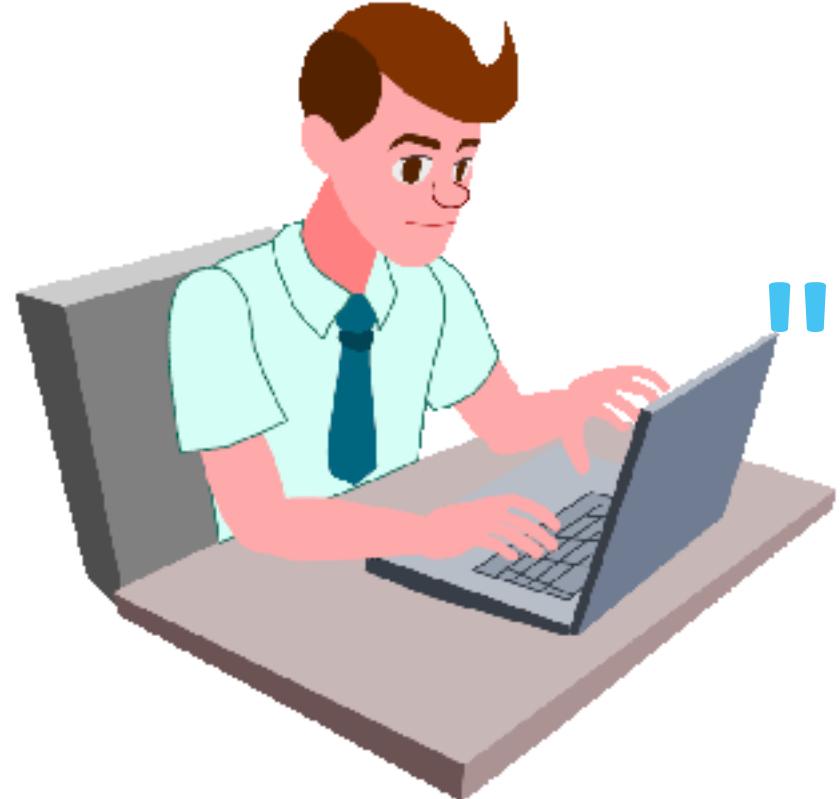
```
[2] ~/grocery (master)$ echo "orange" >>  
shoppingList.txt
```

```
[3] ~/grocery (master)$ git commit -am "Add an orange"  
[master 0e8b5cf] Add an orange 1 file changed, 1  
insertion(+)
```

Git references

- Okay, go on now and take a look at the current repository situation:

```
[4] ~/grocery (master)$ git log --oneline --graph --  
decorate --all  
* 0e8b5cf (HEAD -> master) Add an orange* e4a5e7b  
Add an apple* a57d783 Add a banana to the shopping  
list
```



"Complete Lab 1&2"

Configuring GitLab

Using the Web UI

The following topics will be covered in this lesson:

- Configuring GitLab settings at the instance level
- Configuring GitLab settings at the group level
- Configuring GitLab settings at the project level

Configuring GitLab settings at the instance level

- When you log on to GitLab as an administrator, you will notice a tool icon in the top right of the menu:



Configuring GitLab settings at the instance level

The screenshot shows the GitLab Admin Area Dashboard. The left sidebar contains a navigation menu with the following items:

- Admin Area
- Overview
- Dashboard** (selected)
- Projects
- Users
- Groups
- Jobs
- Runners
- Cohorts
- ConvDev Index
- Monitoring
- Messages
- System Hooks
- Applications
- Abuse Reports (0)
- Deploy Keys
- Service Templates
- Labels
- Appearance
- Settings

The main dashboard area displays the following information:

- Projects: 1** (New project button)
- Users: 1** (New user button)
- Groups: 0** (New group button)
- Statistics** (Forks: 0, Issues: 0, Merge Requests: 0, Notes: 0, Snippets: 0, SSH Keys: 1, Milestones: 0, Active Users: 1)
- Features** (Sign up: green dot, LDAP: grey dot, Gravatar: green dot, OmniAuth: grey dot, Reply by email: grey dot, Container Registry: grey dot, GitLab Pages: grey dot, Shared Runners: green dot)
- Components** (GitLab: 11.1.4 (63daf37), GitLab Shell: 7.1.4, GitLab Workhorse: v5.0.0, GitLab API: v4, Ruby: 2.4.4p296, Rails: 4.2.10, postgresql: 9.6.8, Gitaly Servers: update asap)
- Latest projects** (Administrator / mastering-gitlab..., 2 months ago)
- Latest users** (Administrator, 2 months ago)
- Latest groups**

Messages

- Your GitLab instance has a facility where you can send messages to all of your users.
- These broadcasts can come in handy if you want to inform your users about system-wide events, such as upgrades and scheduled downtime.
- The following is the Admin page, which you can find in the side menu:

The screenshot shows the 'Broadcast Messages' section of the GitLab Admin Area. At the top, there's a note: 'Broadcast messages are displayed for every user.' followed by a pencil icon. Below this is a red input field labeled 'Your message here'. Underneath, there are fields for 'Message' (empty), 'Starts at (UTC)' (set to 2018-11-04 14:57:00 UTC), and 'Ends at (UTC)' (set to 2018-11-04 14:58:00 UTC). A 'Customize colors' link is also present. A button labeled 'Add broadcast message' is located below the scheduling fields. At the bottom, a note says 'After you've scheduled a new message, it can be reused later as well:' followed by a table showing a single message entry: Status: Pending, Preview: 'This is serious message!', Starts: 2018-11-04 14:57:00 UTC, Ends: 2018-11-04 14:58:00 UTC, with edit and delete icons.

Status	Preview	Starts	Ends	Actions
Pending	This is serious message!	2018-11-04 14:57:00 UTC	2018-11-04 14:58:00 UTC	<input type="checkbox"/> <input type="button" value="x"/>

System hooks

- A standard event is raised when you're creating a new project or user.
- Additionally, it can send other types of events as well. Just add a destination URL and (optionally) a secret token:

Edit System Hook

System hooks can be used for binding events when GitLab creates a User or Project.

URL

`http://docker.for.mac.localhost:8081`

Secret Token

`r`

Use this token to validate received payloads

Trigger

System hook will be triggered on set of events like creating project or adding ssh key. But you can also enable extra triggers like Push events.

Repository update events
This URL will be triggered when repository is updated

Push events
This URL will be triggered for each branch updated to the repository

Tag push events
This URL will be triggered when a new tag is pushed to the repository

Merge request events
This URL will be triggered when a merge request is created/updated/merged

SSL verification

Enable SSL verification

Save changes **Test ▾** **Remove**

Plugins

- On this page, you also have the option to configure installed plugins.
- This basically fires a locally installed program instead of calling a URL with parameters.
- It requires you to place the plugin code in `/opt/gitlab/embedded/service/gitlab-rails/plugins`, and it has to be written in a certain way.

Applications

- In this section of the administration page, you have the option to register third-party applications in order to use GitLab as an OAuth authorization provider.
- Open Authorization (OAuth) is an open standard for authorization.
- Users can give a program or website access to their private data that's kept on another website without revealing their username and password.

Abuse reports



joustie

@joustie · Member since November 13, 2018

Overview

Activity

Groups

Contributed projects

Personal projects

Snippets

Personal projects



Report abuse

Abuse reports

Report abuse to GitLab

Please use this form to report users to GitLab who create spam issues, comments or behave inappropriately.

A member of GitLab's abuse team will review your report as soon as possible.

User

joustie (@joustie)



Message

<https://gitlab.joustie.nl:8443/admin/users/joustie>
This is an abuse report!

Explain the problem. If appropriate, provide a link to the relevant issue or comment.

Send report

Abuse reports

Admin Area > Abuse Reports

Abuse Reports

User	Reported by	Message	Action
joustie Joined 1 month ago	Administrator 23 seconds ago	https://gitlab.joustie.nl:8443/admin/users/joustie This is an abuse report!	<button>Remove user & report</button> <button>Block user</button> <button>Remove report</button>

Push rules

- In this section, you can define all kinds of rules that will allow or disallow Git pushes:

Admin Area > Push Rules

Pre-defined push rules.

Rules that define what git pushes are accepted for a project. All newly created projects will use this settings.

Committer restriction

Users can only push commits to this repository that were committed with one of their own verified emails. This

Reject unsigned commits

Only signed commits can be pushed to this repository. This setting will be applied to all projects unless overrid

Do not allow users to remove git tags with `git push`

Tags can still be deleted through the web UI.

Check whether author is a GitLab user

Restrict commits by author (email) to existing GitLab users

Prevent committing secrets to Git

GitLab will reject any files that are likely to contain secrets. The list of file names we reject is available in the [do](#)

Deploy Keys

- In this section, you can register SSH keys, which are known as Global Shared Deploy keys.
- They allow read-only or read-write (if enabled) access to be configured on any repository in the entire GitLab installation.
- When the administrator has registered them here, you can assign them in your project, as shown in the following screenshot:

Deploy Keys

Collapse

Deploy keys allow read-only or read-write (if enabled) access to your repository. Deploy keys can be used for CI, staging or production servers. You can create a deploy key or add an existing one.

Create a new deploy key for this project

Title

Key

Paste a machine public key here. Read more about how to generate it [here](#)

Write access allowed

Allow this key to push to repository as well? (Default only allows pull access.)

Add key

Enabled deploy keys 1 Privately accessible deploy keys 0 Publicly accessible deploy keys 0

Deploy key

Project usage

Created

Test

f4:cc:ec:76:d0:1c:86:1d:45:6d:a7:6e:b3:df:32:7c

Current project

1 minute ago



Appearance

- You can define some cosmetic aspects of your GitLab instance on the Appearance settings page:

Admin Area > Appearance

Appearance settings

You can modify the look and feel of GitLab here

Navigation bar:

Header logo

no file selected

Maximum file size is 1MB. Pages are optimized for a 28px tall header logo

Favicon:

Favicon

no file selected

Maximum file size is 1MB. Image size must be 32x32px. Allowed image formats are '.png' and '.ico'.
Images with incorrect dimensions are not resized automatically, and may result in unexpected behavior.

Sign in/Sign up pages:

Title

Description

Description parsed with *GitLab Flavored Markdown*.

Logo

no file selected

Maximum file size is 1MB. Pages are optimized for a 640x360 px logo.

New project pages:

New project guidelines

Guidelines parsed with *GitLab Flavored Markdown*.

Save

Appearance

- After you have done this, log out. You will be redirected to the front page:

GitLab Community Edition



Open source software to collaborate on code

Manage Git repositories with fine-grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.

Sign in	Register
Username or email <input type="text"/>	
Password <input type="password"/>	
<input type="checkbox"/> Remember me	Forgot your password?
Sign in	

Didn't receive a confirmation email? [Request a new one.](#)

Sign-in restrictions

- It is possible to use a hardware token device as a second factor, as you can see in the following screenshot (this only works in Chrome):



Sign-in restrictions

- You can also choose to use a code generator app such as Google Authentication:

Two-Factor Authentication

Two-factor authentication code

Enter the code from the two-factor app on your mobile device. If you've lost your device, you may enter one of your recovery codes.

Verify code

Elasticsearch

- Elasticsearch indexing
- Use the [new repository indexer \(beta\)](#)
- Search with Elasticsearch enabled

URL

`http://localhost:9200`

The url to use for connecting to Elasticsearch. Use a comma-separated list to support clustering (e.g., "http://localhost:9200, http://localhost:9201").

Number of Elasticsearch shards

5



How many shards to split the Elasticsearch index over. Changes won't take place until the index is [recreated](#).

Number of Elasticsearch replicas

1



How many replicas each Elasticsearch shard has. Changes won't take place until the index is [recreated](#).

Elasticsearch

- You can also limit what will be indexed:

Elasticsearch indexing restrictions

- Limit namespaces and projects that can be indexed

Elasticsearch

- Another option is to connect to an Elasticsearch instance that you are running in the Amazon cloud.
- You can specify connection settings here as well if you have this set up:

Elasticsearch AWS IAM credentials

Using AWS hosted Elasticsearch with IAM credentials

AWS region

us-east-1

Region that elasticsearch is configured

AWS Access Key

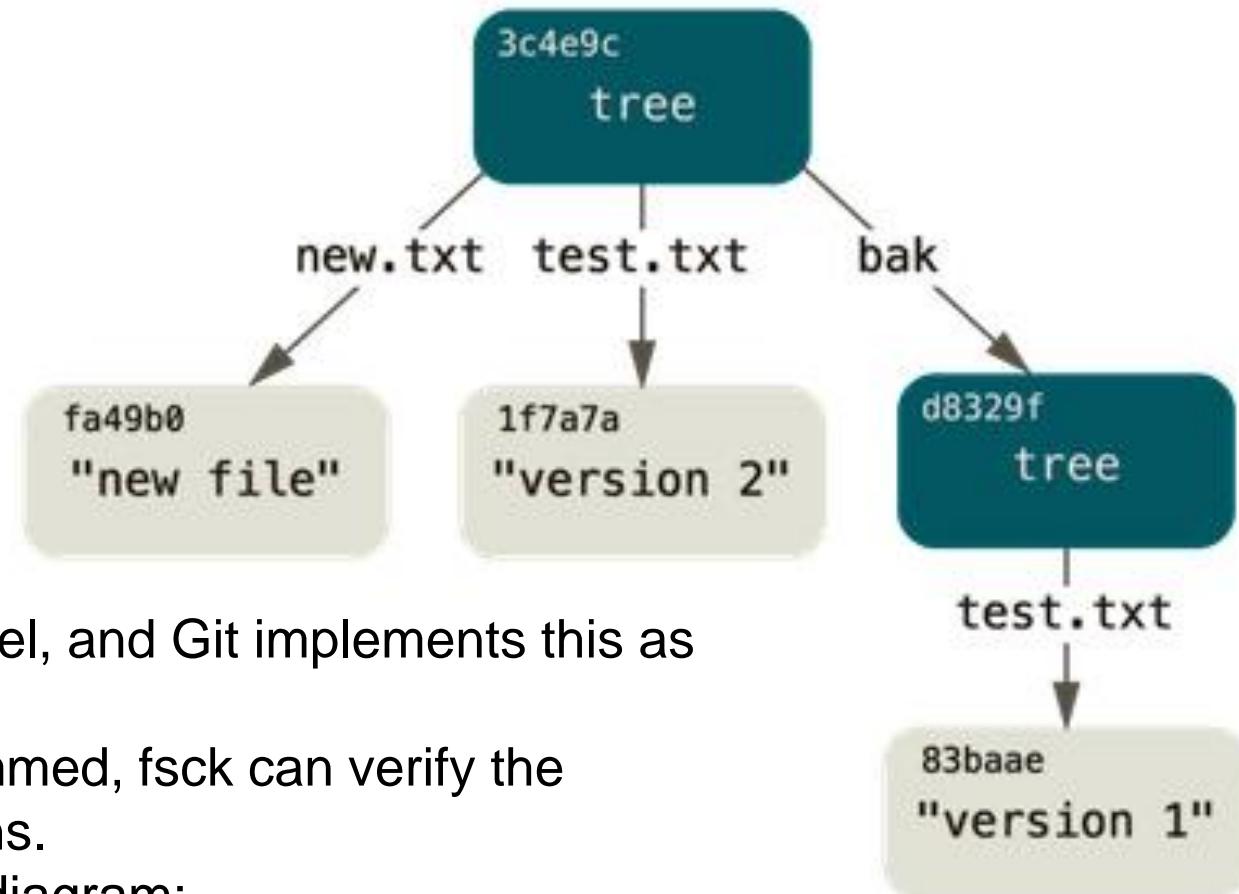
AWS Access Key. Only required if not using role instance credentials

AWS Secret Access Key



AWS Secret Access Key. Only required if not using role instance credentials

Repository maintenance



- A filesystem is classed as a graph model, and Git implements this as tree and blob objects.
- Because all of the items are check-summed, fsck can verify the integrity of the objects and their relations.
- This graph is depicted in the following diagram:

Auto DevOps settings

Continuous Integration and Deployment

Auto DevOps, runners and job artifacts

- Default to Auto DevOps pipeline for all projects

The Auto DevOps pipeline will run if no alternative CI configuration file is found. [More information](#)

Auto devops domain

domain.com

Specify a domain to use by default for every project's Auto Review Apps and Auto Deploy stages.

[Collapse](#)

Help page

Help page

- You can also customize the way the Help page for GitLab is presented.
- There's the option to provide some custom text, which will be displayed on top of the Help page:

Help page text and support page url.

Help page text

This is a help page.

Markdown enabled

Hide marketing-related entries from help

Support page URL

<http://blog.joustie.nl>

Alternate support URL for help page

[Save changes](#)

Help page

- The following is a screenshot of the standard Help page:

Help > Help

GitLab Community Edition 11.4.5 update asap

GitLab is open source software to collaborate on code.

Manage git repositories with fine-grained access controls that keep your code secure.

Perform code reviews and enhance collaboration with merge requests.

Each project can also have an issue tracker and a wiki.

Used by more than 100,000 organizations, GitLab is the most popular solution to manage git repositories on-premises.

Read more about GitLab at about.gitlab.com.

[Check the current instance configuration](#)

GitLab Documentation

Welcome to [GitLab](#), a Git-based fully featured platform for software development!

GitLab offers the most scalable Git-based fully integrated platform for software development, with flexible products and subscriptions. To understand what features you have access to, check the [GitLab subscriptions](#) below.

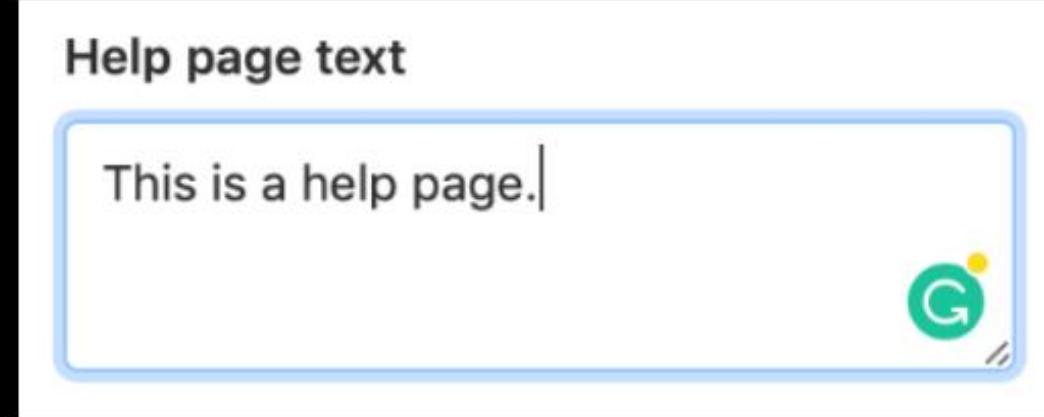
Quick help

[See our website for getting help](#)

[Use the search bar on the top of this page](#)

Help page

Let's make some changes by adding **This is a help page.**:



The result will be as follows:

A screenshot of a GitLab Community Edition 11.4.5 help page. The top navigation bar shows "Help > Help" and the page content area displays "This is a help page.". Below this, the main content area features the heading "GitLab Community Edition 11.4.5" with a "update asap" button, followed by "GitLab Documentation" and a welcome message. A sidebar on the right is titled "Quick help" and contains links: "See our website for getting help", "Use the search bar on the top of this page", and "Use shortcuts". At the bottom, there are buttons for "General documentation" and "GitLab CI/CD docs".

Pages

- If you use the GitLab Pages feature, you can specify the maximum size of pages.
- You can set it to zero if you want the size to be unlimited.
- You can also allow users to prove that they own a domain before you serve a page for it:

Pages

[Collapse](#)

Size and domain settings for static websites

Maximum size of pages (MB)

100



0 for unlimited

Require users to prove ownership of custom domains

Domain verification is an essential security measure for public GitLab sites. Users are required to demonstrate they control a domain before it is enabled [?](#)

Localization

- This is a big section of the settings since there are many localization settings for software products.
- The only one that is exposed in this screen is Default first day of the week:

Localization

Various localization settings.



Collapse

Default first day of the week

Monday

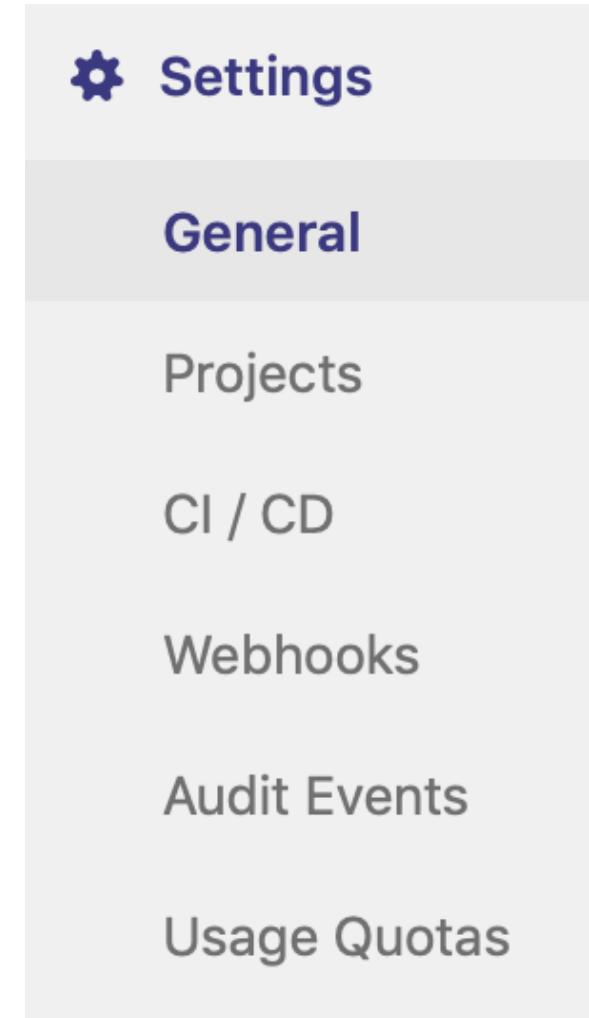


Default first day of the week in calendars and date pickers.

Save changes

Configuring GitLab settings at the group level

- You will see a submenu with the items that you can configure.
- It looks a lot like the UI in the admin area but is, of course, scoped to the group:



Configuring GitLab settings at the group level

Restrict access by IP address

192.168.1.0/24

This group, including all subgroups, projects and git repositories, will only be reachable from the specified IP address range.

Example: [192.168.0.0/24](#). [Read more](#).

Large File Storage

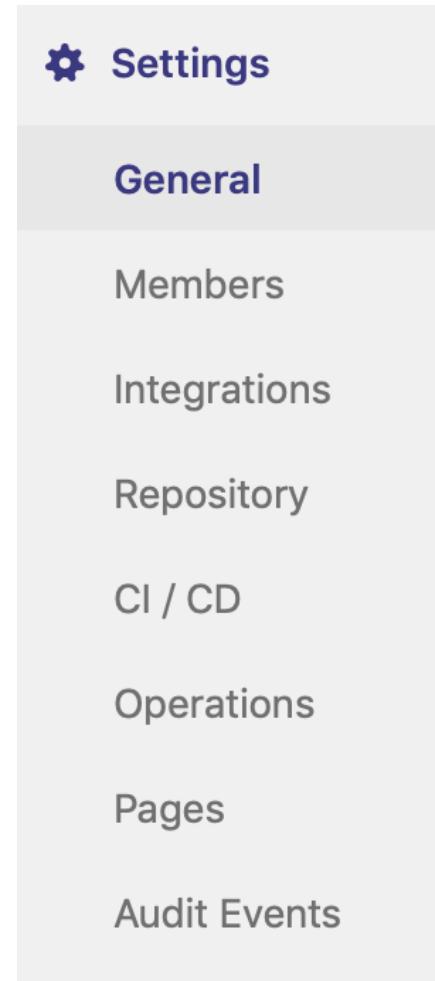
Check the [documentation](#).

- Allow projects within this group to use Git LFS
This setting can be overridden in each project.

- In the following screenshot, you can see that 192.168.1.0/24 is the only IP range that's allowed to see the group content:

Configuring GitLab settings at the project level

- There are also options for setting an individual project.
- If you browse to one of your projects, you will see a Settings menu on the left:



Naming, topics, avatar

- Under the General settings, you can find the fields:

Naming, topics, avatar

Update your project name, topics, description and avatar.

Project name

Project ID

Topics

Separate topics with commas.

Project description (optional)

Repository size limit (MB)

 ↑ ↓

The total size of this project's repository including files in LFS will be limited to this size. 0 for unlimited. Leave empty to inherit the group/global value.

Project avatar



No file chosen

The maximum file size allowed is 200KB.

Visibility, project features, permissions

Visibility, project features, permissions

Choose visibility level, enable/disable project features (issues, repository, wiki, snippets) and set permissions.

Project visibility 

Private 

The project is accessible only by members of the project. Access must be granted explicitly to each user.

Issues

Lightweight issue tracking system for this project

 Only Project Members 

Repository

View and edit files in this project

 Only Project Members 

Merge requests

Submit changes to be merged upstream

 Only Project Members 

Merge requests

- For every executed merge request, there is a Git session on the server running the same Git binary that you have on your workstation.
- For instance, you can specify that the server side never does a merge commit:

Merge requests

Collapse

Choose your merge method, options, checks, and set up a default merge request description template.

Merge method

This will dictate the commit history when you merge a merge request

Merge commit

Every merge creates a merge commit

Merge commit with semi-linear history

Every merge creates a merge commit

Fast-forward merges only

When conflicts arise the user is given the option to rebase

Fast-forward merge

No merge commits are created

Fast-forward merges only

When conflicts arise the user is given the option to rebase

Merge options

Additional merge request capabilities that influence how and when merges will be performed

Merge pipelines will try to validate the post-merge result prior to merging

Pipelines need to be configured to enable this feature. [?](#)

Allow merge trains

Automatically resolve merge request diff discussions when they become outdated

Show link to create/view merge request when pushing from the command line

Merge checks

These checks must pass before merge requests can be merged

Pipelines must succeed

Pipelines need to be configured to enable this feature. [?](#)

All discussions must be resolved

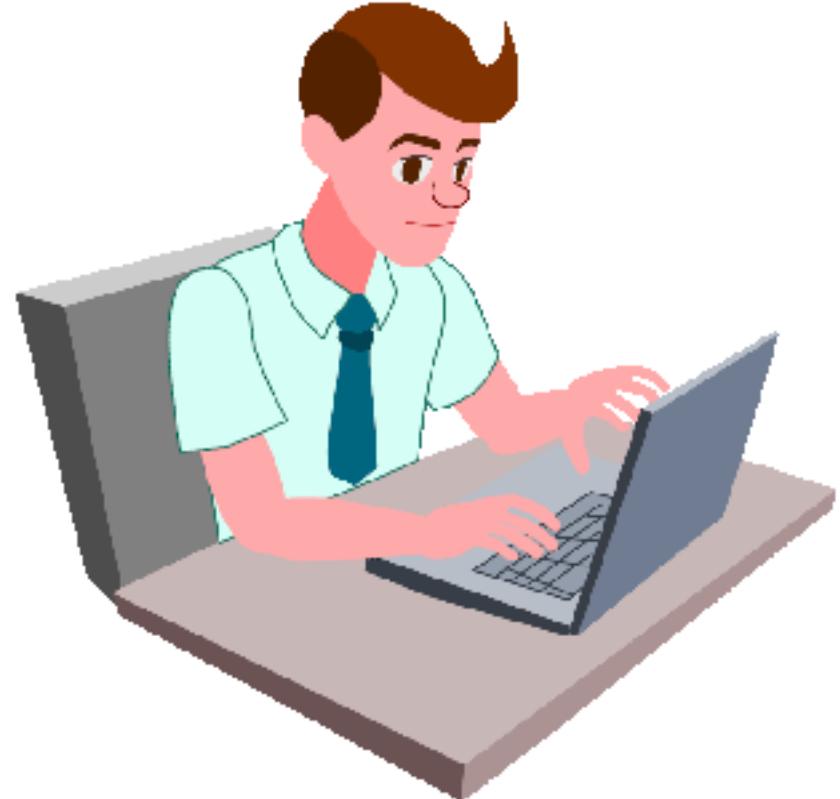
Default description template for merge requests [?](#)

Description parsed with [GitLab Flavored Markdown](#)

Save changes

Summary

- In this lesson, we discussed how to configure an existing GitLab application instance via the web interface.
- The administration pages of GitLab give you a lot of control over your instance.
- After going through those pages, we explained the various items that can be managed.



"Complete Lab 3"

3. Branching



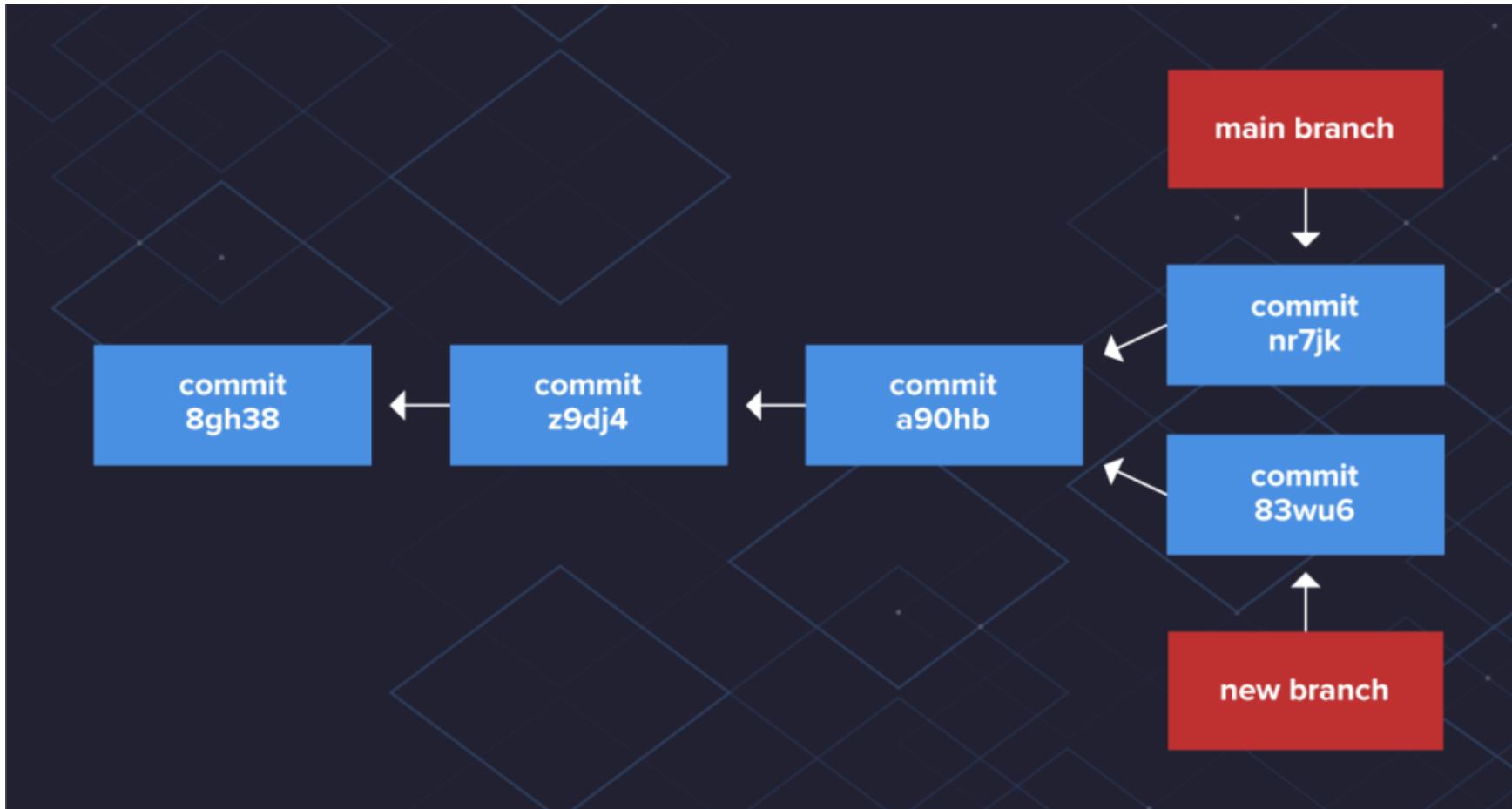
Git branching

- Branches are not just exclusive to Git. However, in this course we focus on Git due to the many advantages this model of branching offers.
- Consequently, before we delve into the various branching strategies out there, including Git branching strategies.
- Put simply, Git and other version control tools allow developers to track, manage and organize their code.

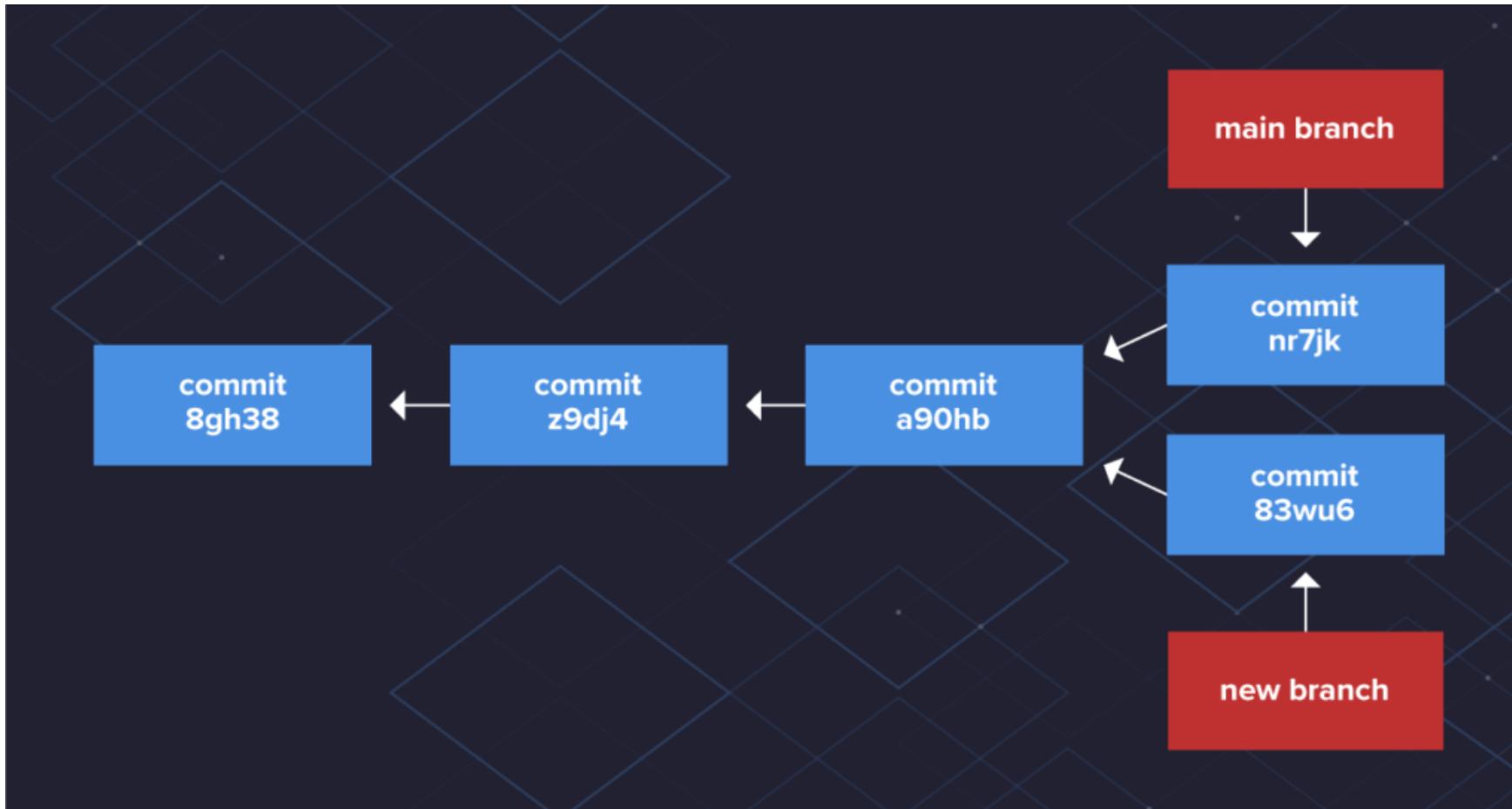
Git Branching



Git Branching



Git Branching



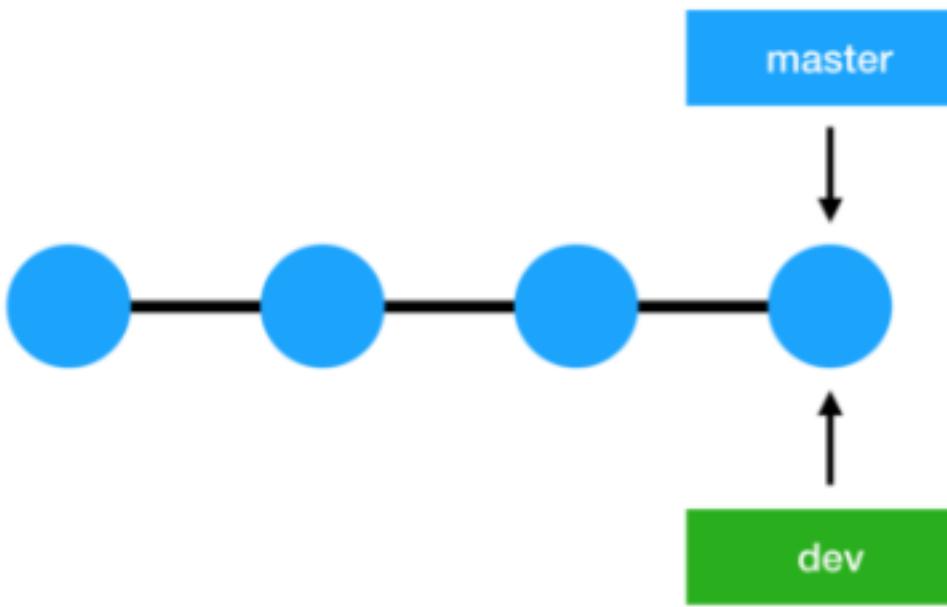
Branch Naming Strategies

- Branch names can be anything you'd like.
- Your organization or project may have standards outlined for branch naming.
- For example, naming the branch based on the person responsible for working on the branch and a description or work item:
 - username/description
 - username/workitem

Branch Naming Strategies

- You can name a branch to indicate the branch's function, like a feature, bug fix, or hotfix:
 - bugfix/description
 - feature/feature-name
 - hotfix/description

Introduction



Introduction

- If you are developing a small application in a big corporation as a developer.
- Or you are trying to wrap your head around an open source project from GitHub, you have already been using branches with Git.

Managing your local branches

- Suppose you are just having your local Git repository, and you have no intentions at the moment to share the code with others.
- However, you can easily share this knowledge while working with a repository with one or more remotes.

Getting ready

- Use the following command to clone the jgit repository to match:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit
```

How to do it...

- Whenever you start working on a bug fix or a new feature in your project, you should create a branch.
- You can do so using the following code:

```
$ git branch newBugFix  
$ git branch  
* master  
newBugFix
```

How to do it...

- The newBugFix branch points to the current HEAD I was on at the time of the creation.
- You can see the HEAD with git log -1:

```
$ git log -1 newBugFix --format=format:%H  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

- If you want to add a description to the branch, you can do it with the --edit-description option for the git branch command:

```
$ git branch --edit-description newBugFix
```

How to do it...

- The previous command will open an editor where you can type in a description:

Refactoring the Hydro controller

The hydro controller code is currently horrible needs to be refactored.

- Close the editor and the message will be saved.

How it works...

- Git stores the information in the local git config file; this also means that you cannot push this information to a remote repository.
- To retrieve the description for the branch, you can use the --get flag for the git config command:

```
$ git config --get branch.newBugFix.description
```

Refactoring the Hydro controller

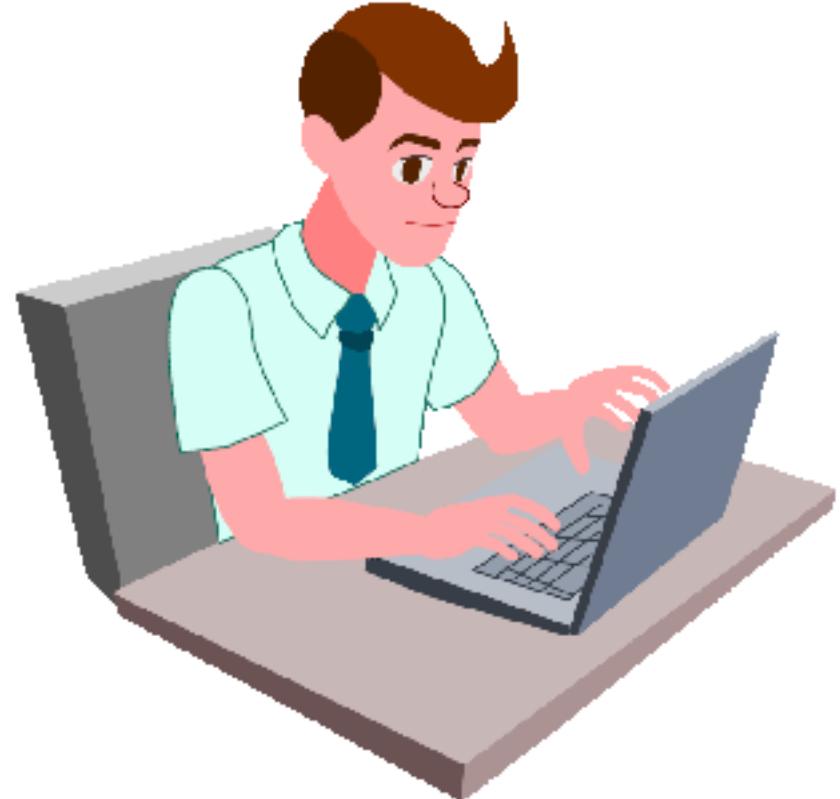
The hydro controller code is currently horrible needs to be refactored.

How it works...

- The branch information is stored as a file in .git/refs/heads/newBugFix:

```
$ cat .git/refs/heads/newBugFix  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

- Note that it is the same commit hash we retrieved with the git log command



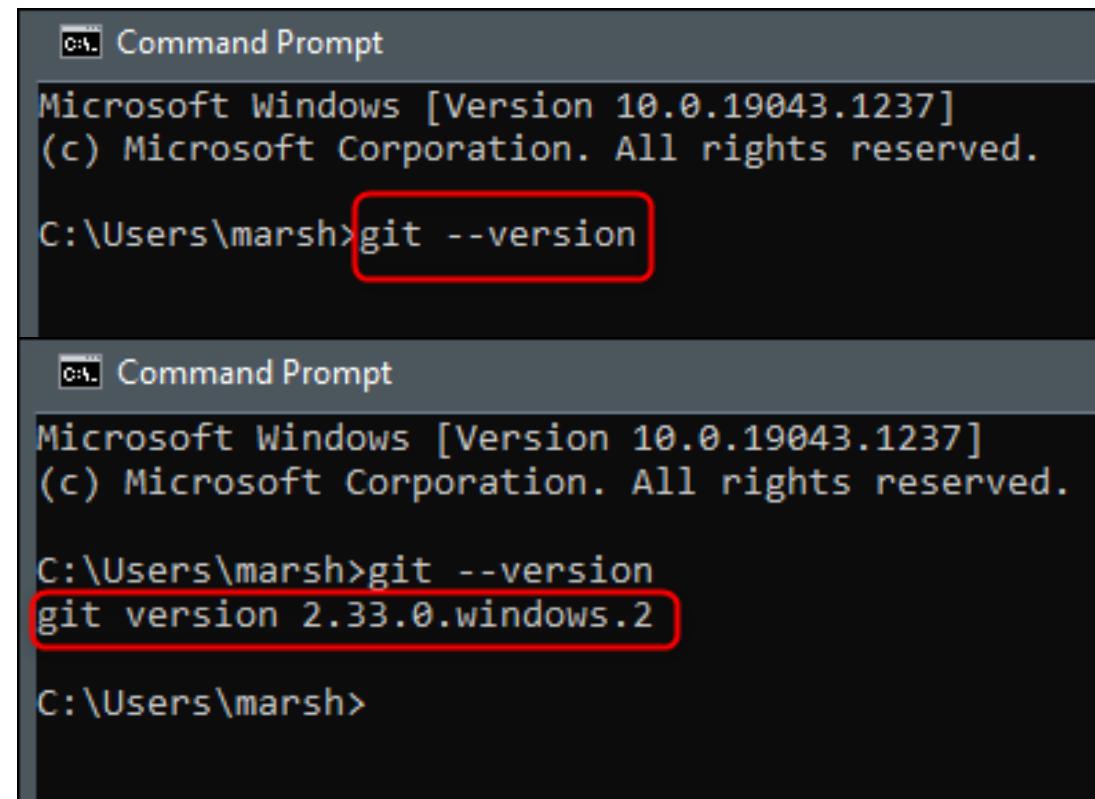
"Complete Lab 4"

4. Configuring Git



Checking your Git version

- Once open, run this command:
git --version



The image shows two separate Command Prompt windows side-by-side. Both windows have a dark gray header bar with the text "Command Prompt". Below the header, both windows display identical system information:

```
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.
```

In the first window, the command `git --version` is entered at the prompt, and its output is shown in red:

```
C:\Users\marsh>git --version
```

In the second window, the command `git --version` is also entered at the prompt, and its output is shown in red:

```
git version 2.33.0.windows.2
```

Both windows end with a blank command line prompt:

```
C:\Users\marsh>
```

Git Configuration Levels

- In Git different layers that can be configured. The layers are:

SYSTEM: This layer is system-wide and found in
`/etc/gitconfig`

GLOBAL: This layer is global for the user and found in
`~/.gitconfig`

LOCAL: This layer is local to the current repository and found
in `.git/config`

Viewing your configurations

- You can use:

`git config --list`

- or look at your `~/.gitconfig` file. The local configuration will be in your repository's `.git/config` file.

Use:

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

to see where that setting is defined (global, user, repo, etc...)

Configuring your username and email

```
# Create a project specific config, you have to execute  
this under the project's directory.  
$ git config user.name "John Doe"
```

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

5. Rebasing



Rebasing

Basically, with git rebase you rewrite history; with this statement, I mean you can use rebase command to achieve the following:

- Combine two or more commits into a new one
- Discard a previous commit you did
- Change the starting point of a branch, split it, and much more

Rebasing

Reassembling commits

- Suppose we erroneously added half a grape in the shoppingList.txt file, then the other half, but at the end we want to have only one commit for the entire grape; follow me with these steps.
- Add a gr to the shopping list file:

```
[1] ~/grocery (master)  
$ echo -n "gr" >> shoppingList.txt
```

Rebasing

- The -n option is for not adding a new line.
- Cat the file to be sure:

```
[2] ~/grocery (master)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

```
peach
```

```
gr
```

Rebasing

- Let's suppose, we have a commit with half a grape.
- Go on and add the other half, ape:

```
[4] ~/grocery (master)  
$ echo -n "ape" >> shoppingList.txt
```

- Check the file:

```
[5] ~/grocery (master)  
$ cat shoppingList.txt
```

banana

apple

orange

peach

grape



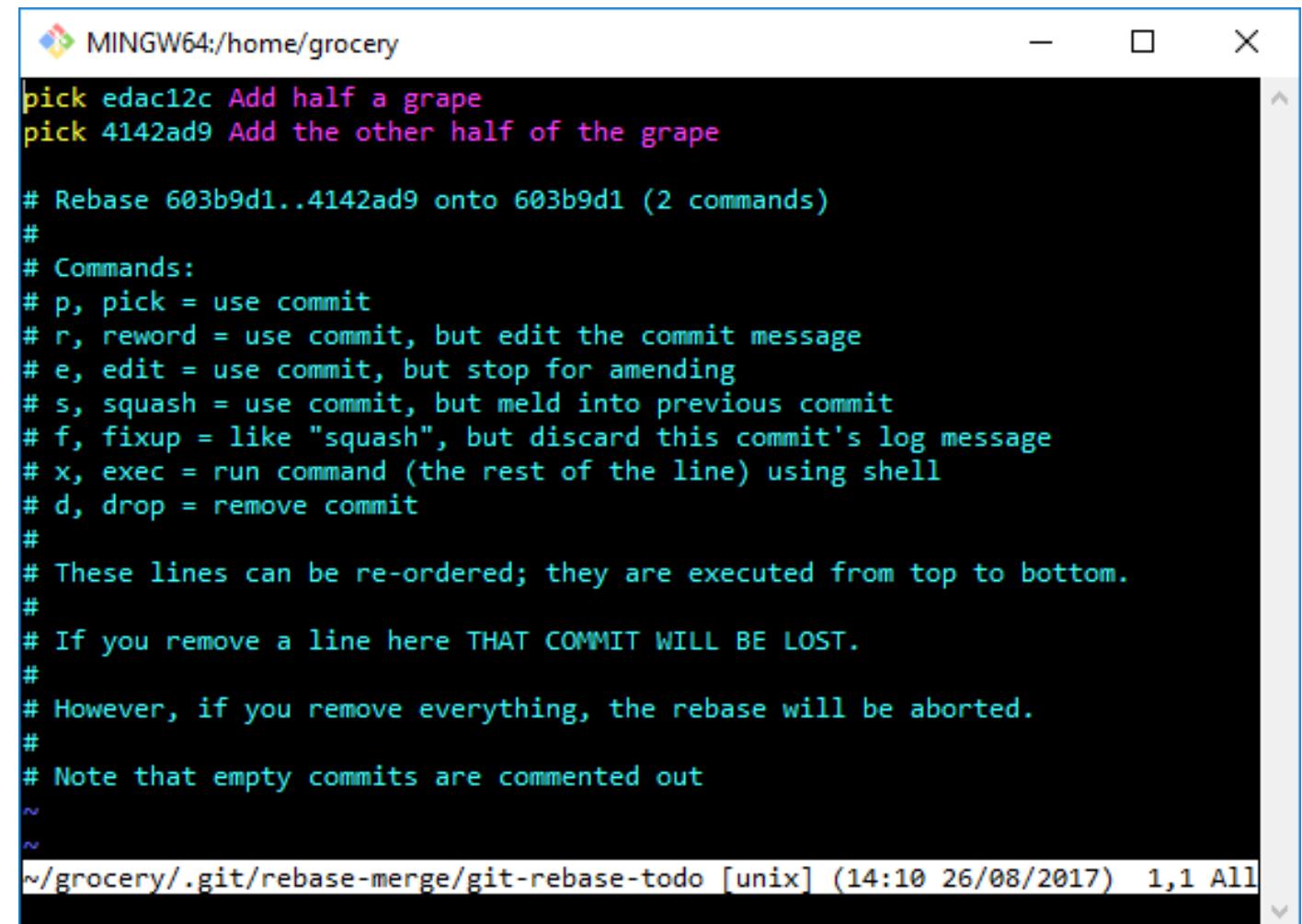
Rebasing

- Check the log:

```
[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 4142ad9 (HEAD -> master) Add the other half of the grape
* edac12c Add half a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Rebasing

This is a screenshot
of the console:



The screenshot shows a terminal window titled "MINGW64:/home/grocery". The window contains a list of git rebase commands. At the top, two commits are listed: "pick edac12c Add half a grape" and "pick 4142ad9 Add the other half of the grape". Below these, a series of comments explain the rebase process, including the meaning of "p", "r", "e", "s", "f", "x", and "d" operations, the execution of shell commands, and the consequences of removing lines. The text ends with a note about empty commits and a timestamp at the bottom: "/grocery/.git/rebase-merge/git-rebase-todo [unix] (14:10 26/08/2017) 1,1 All".

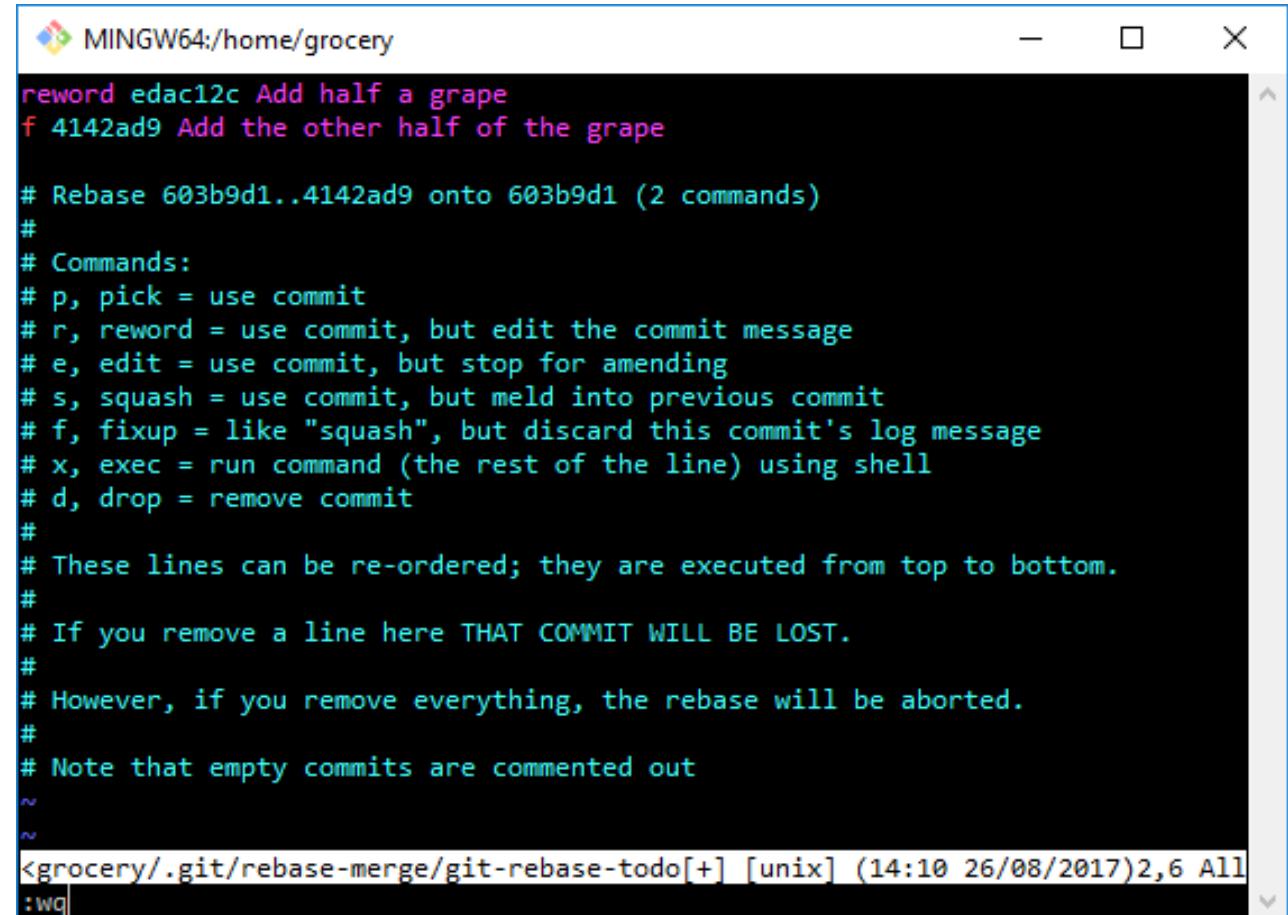
```
pick edac12c Add half a grape
pick 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~

~/grocery/.git/rebase-merge/git-rebase-todo [unix] (14:10 26/08/2017) 1,1 All
```

Rebasing

- To resolve our issue, I will reword the first commit and then fixup the second; the following is a screenshot of my console:



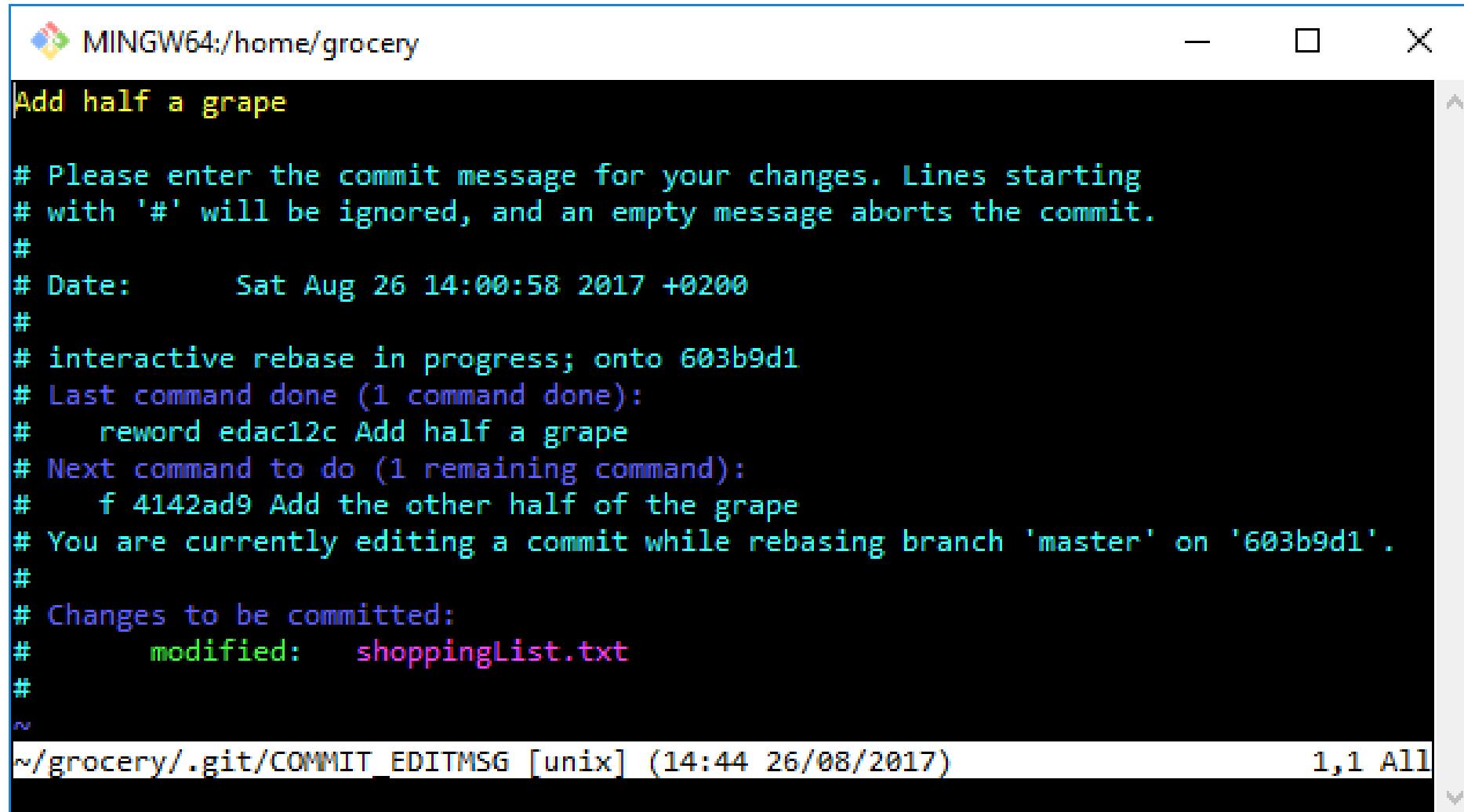
The screenshot shows a terminal window titled "MINGW64:/home/grocery". The command entered is:

```
reword edac12c Add half a grape
f 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~

<grocery/.git/rebase-merge/git-rebase-todo[+] [unix] (14:10 26/08/2017)2,6 All
:wq|
```

Rebasing



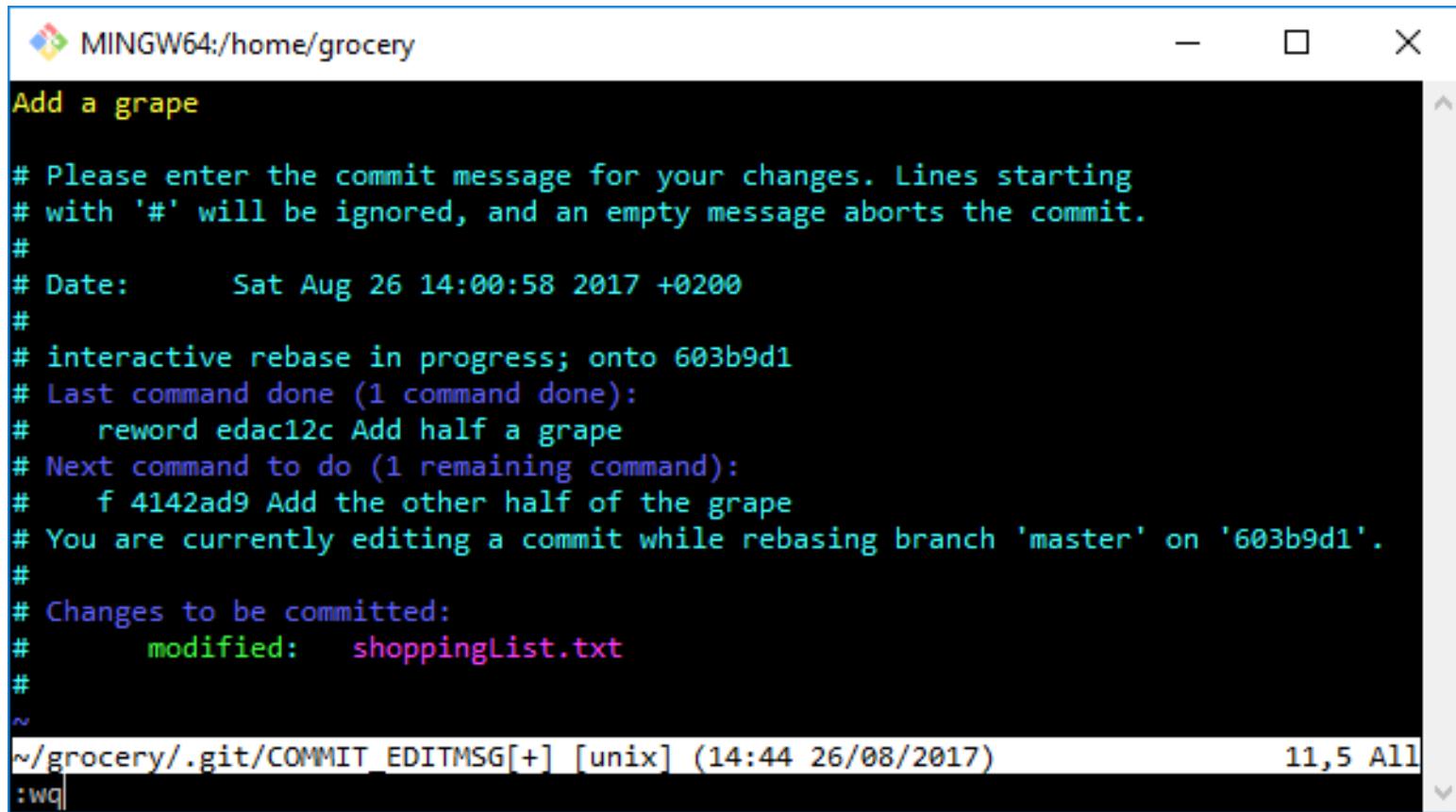
The screenshot shows a terminal window titled "MINGW64:/home/grocery". The window contains a commit message for a rebasing operation. The message starts with "Add half a grape" followed by a series of commit log entries. These entries include the date ("Sat Aug 26 14:00:58 2017 +0200"), the command being run ("interactive rebase in progress; onto 603b9d1"), the last command done ("reword edac12c Add half a grape"), the next command to do ("f 4142ad9 Add the other half of the grape"), and the current branch being rebased ("You are currently editing a commit while rebasing branch 'master' on '603b9d1'"). The message concludes with a note about changes to be committed ("Changes to be committed") and a list of modified files ("modified: shoppingList.txt"). The bottom status bar of the terminal window displays the path "~/.git/COMMIT_EDITMSG [unix] (14:44 26/08/2017)" and the file count "1,1 All".

```
Add half a grape

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 26 14:00:58 2017 +0200
#
# interactive rebase in progress; onto 603b9d1
# Last command done (1 command done):
#   reword edac12c Add half a grape
# Next command to do (1 remaining command):
#   f 4142ad9 Add the other half of the grape
# You are currently editing a commit while rebasing branch 'master' on '603b9d1'.
#
# Changes to be committed:
#   modified:   shoppingList.txt
#
~/.git/COMMIT_EDITMSG [unix] (14:44 26/08/2017) 1,1 All
```

Rebasing

- Now edit the message, and then save and exit, like in the following screenshot:



The screenshot shows a terminal window titled "MINGW64:/home/grocery". The window contains the following text:

```
Add a grape

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 26 14:00:58 2017 +0200
#
# interactive rebase in progress; onto 603b9d1
# Last command done (1 command done):
#   reword edac12c Add half a grape
# Next command to do (1 remaining command):
#   f 4142ad9 Add the other half of the grape
# You are currently editing a commit while rebasing branch 'master' on '603b9d1'.
#
# Changes to be committed:
#       modified:    shoppingList.txt
#
~

~/grocery/.git/COMMIT_EDITMSG[+] [unix] (14:44 26/08/2017)           11,5 All
:wq
```

The bottom of the terminal window shows the command `:wq` being typed in the input field.

Rebasing

- This is the final message from Git:

```
[8] ~/grocery (master)
$ git rebase -i HEAD~2
unix2dos: converting file C:/Users/san/Google Drive/Packt/PortableGit/home/grocery/[detached HEAD
53c73dd] Add a grape
Date: Sat Aug 26 14:00:58 2017 +0200
1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/master.
```

- Take a look at the log:

```
[9] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Rebasing

- Let's start by creating a new branch that points to commit 0e8b5cf, the orange one:

```
[1] ~/grocery (master)  
$ git branch nuts 0e8b5cf
```

- This time I used the git branch command followed by two arguments, the name of the branch and the commit where to stick the label.
- As a result, a new nuts branch has been created:

```
[2] ~/grocery (master)  
$ git log --oneline --graph --decorate --all  
* 6409527 (HEAD -> master) Add a grape  
* 603b9d1 Add a peach  
| * a8c6219 (melons) Add a watermelon  
| * ef6c382 (berries) Add a blackberry  
|/  
* 0e8b5cf (nuts) Add an orange  
* e4a5e7b Add an apple  
* a57d783 Add a banana to the shopping list
```

Rebasing

- Move HEAD to the new branch with the git checkout command:

```
[3] ~/grocery (master)
```

```
$ git checkout nuts
```

```
Switched to branch 'nuts'
```

- Okay, now it's time to add a walnut; add it to the shoppingList.txt file:

```
[4] ~/grocery (nuts)
```

```
$ echo "walnut" >> shoppingList.txt
```

- Then do the commit:

```
[5] ~/grocery (nuts)
```

```
$ git commit -am "Add a walnut"
```

```
[master 3d3ae9c] Add a walnut
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

Rebasing

Check the log:

```
[6] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 9a52383 (HEAD -> nuts) Add a walnut
| * 6409527 (master) Add a grape
| * 603b9d1 Add a peach
|/
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Rebasing

- Let's do it, rebasing the nuts branch on top of master; double-check that you actually are in the nuts branch, as a rebase command basically rebases the current branch (nuts) to the target one, master; so:

```
[7] ~/grocery (nuts)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add a walnut
Using index info to reconstruct a base tree...
M  shoppingList.txt
Falling back to patching base and 3-way merge...
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Patch failed at 0001 Add a walnut
The copy of the patch that failed is found in: .git/rebase-apply/patch

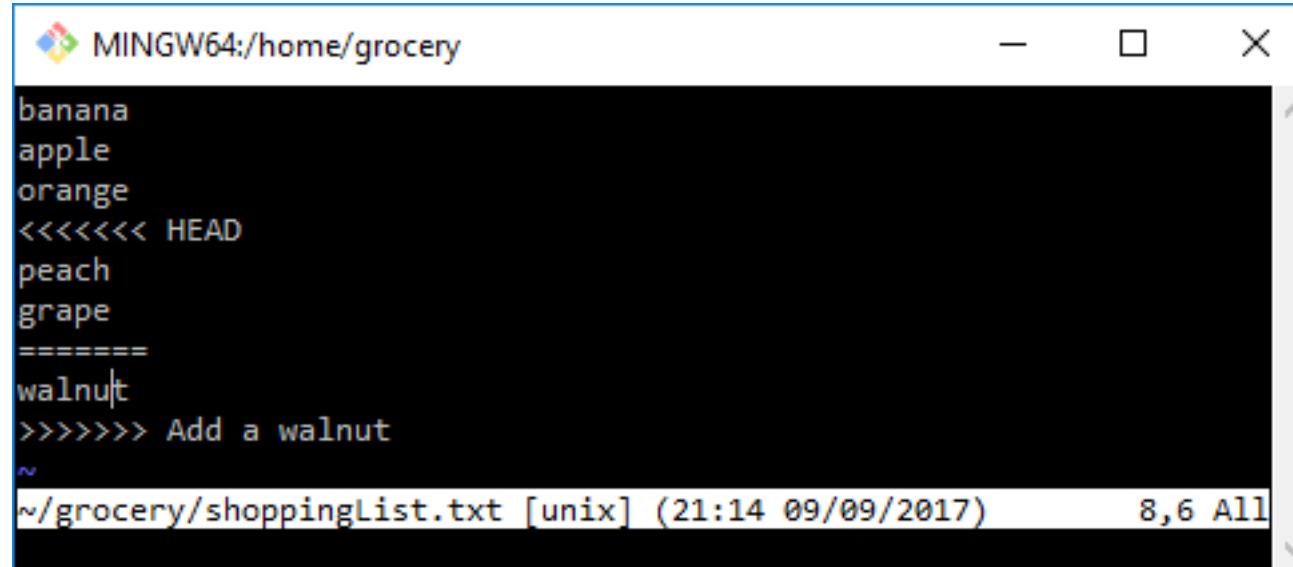
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".

error: Failed to merge in the changes.
```

Rebasing

- Now, back to our repository; if you open the file with Vim, you can see the generated conflict:

```
[8] ~/grocery (nuts|REBASE 1/1)
$ vi shoppingList.txt
```



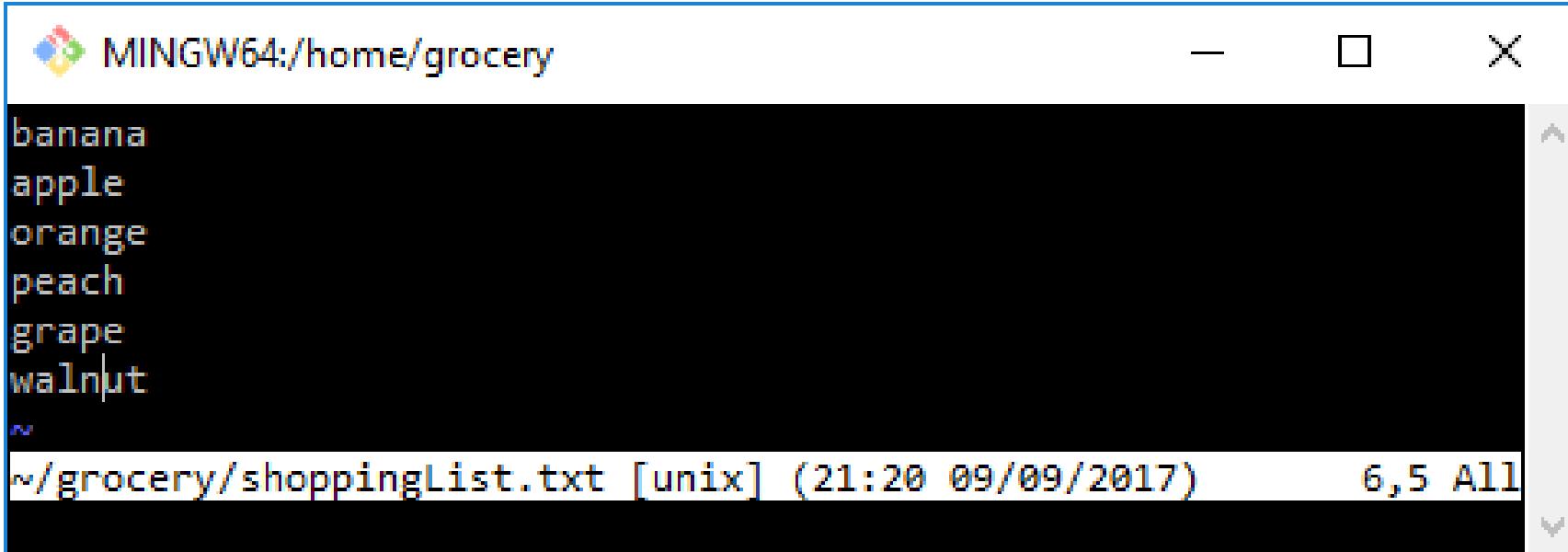
The screenshot shows a terminal window titled "MINGW64:/home/grocery". Inside the terminal, the file "shoppingList.txt" is being edited in Vim. The content of the file is as follows:

```
banana
apple
orange
<<<<< HEAD
peach
grape
=====
walnut
>>>> Add a walnut
~
```

A conflict is visible between the current state (HEAD) and the rebase operation. The word "walnut" is preceded by "<<<<< HEAD" and followed by ">>>> Add a walnut". A cursor is positioned at the end of "walnut". The status bar at the bottom of the terminal window indicates the file path as "~/grocery/shoppingList.txt [unix]" and the date/time as "(21:14 09/09/2017)". It also shows "8,6 All" which likely refers to the number of changes and the current status.

Rebasing

- I will fix it adding the walnut at the end of the file:



A screenshot of a terminal window titled "MINGW64:/home/grocery". The window contains a list of fruits: banana, apple, orange, peach, grape, and walnut. The word "walnut" is followed by a cursor. Below the list, there is a tilde (~) and the path "/grocery/shoppingList.txt [unix] (21:20 09/09/2017)". The status bar at the bottom right shows "6,5 All".

Rebasing

- Now, the next step is to git add the shoppingList.txt file to the staging area, and then go on with the git rebase --continue command, as the previous message suggested:

```
[9] ~/grocery (nuts|REBASE 1/1)
```

```
$ git add shoppingList.txt
```

```
[10] ~/grocery (nuts|REBASE 1/1)
```

```
$ git rebase --continue
```

```
Applying: Add a walnut
```

```
[11] ~/grocery (nuts)
```

```
$
```

Rebasing

Now take a look at the repo using git log as usual:

```
[12] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 383d95d (HEAD -> nuts) Add a walnut
* 6409527 (master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

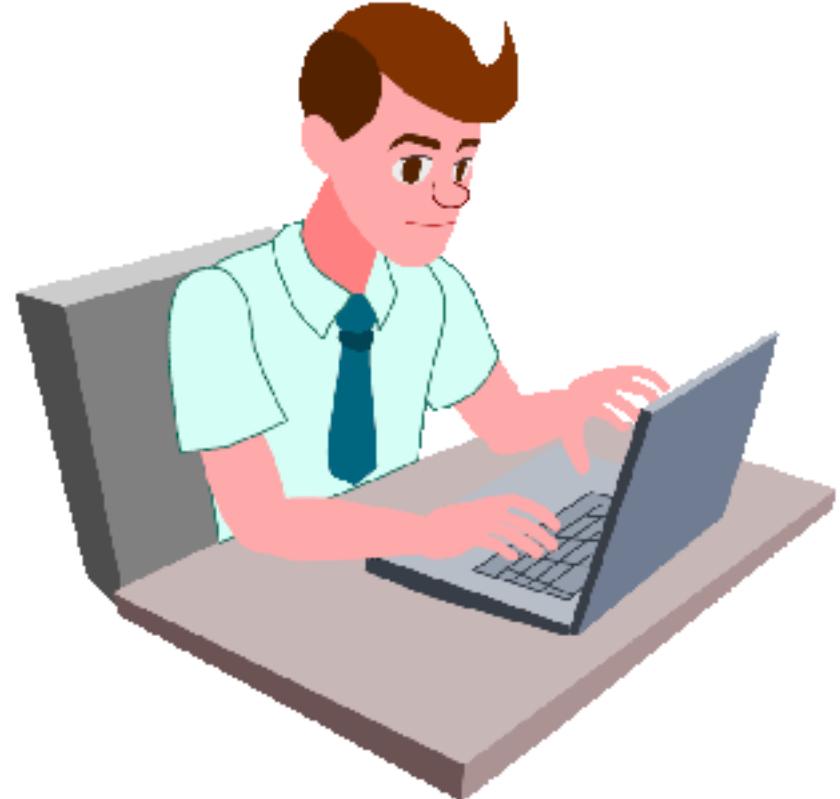
Rebasing

- Okay, now to keep the simplest and most compact repository.
- We cancel the walnut commit and put everything back in place as it was before this little experiment, even removing the nuts branch:

```
[13] ~/grocery (nuts)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[14] ~/grocery (nuts)
$ git checkout master
Switched to branch 'master'

[15] ~/grocery (master)
$ git branch -d nuts
Deleted branch nuts (was 6409527).
```



"Complete Lab 5"

6. Merging



Merging branches

In Git, merging two (or more!) branches is the act of making their personal history meet each other. When they meet, two things can happen:

- Files in their tip commit are different, so some conflict will rise
- Files do not conflict
- Commits of the target branch are directly behind commits of the branch we are merging, so a fast-forward will happen

Merging branches

- Let's give it a try.
- We can try to merge the melons branch into the master one; to do so, you have to check out the target branch, master in this case, and then fire a git merge <branch name> command; as I'm already on the master branch, I go straight with the merge command:

```
[1] ~/grocery (master)
$ git merge melons
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merging branches

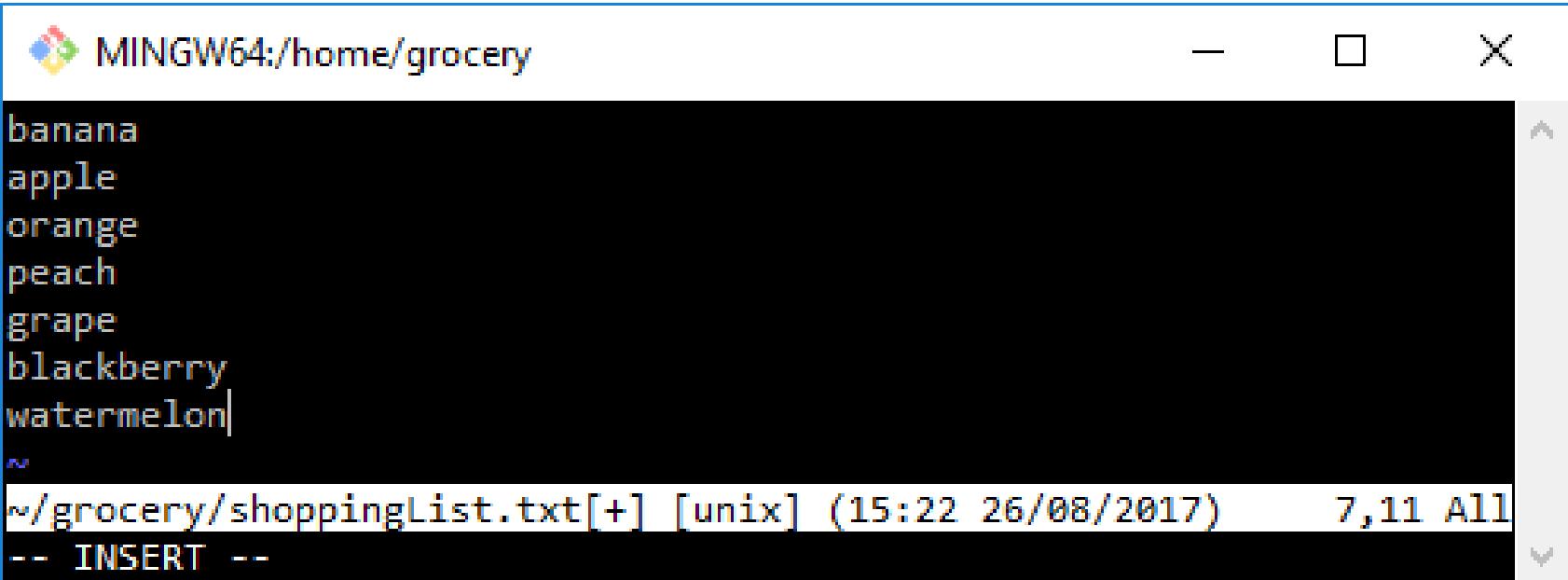
- See the conflict with git diff:

```
[2] ~/grocery (master|MERGING)
$ git diff
diff --cc shoppingList.txt
index 862debc,7786024..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,5 +1,10 @@@
    banana
    apple
    orange
+<<<<<< HEAD
    +peach
    - grape

    ++grape
    ++++++
    + blackberry
    + watermelon
    +>>>>> melons
```

Merging branches

- I will edit the file enqueueing blackberry and watermelon after peach and grape, as per the following screenshot:



The screenshot shows a terminal window titled "MINGW64:/home/grocery". The window contains a list of fruits in a file named "shoppingList.txt". The list includes "banana", "apple", "orange", "peach", "grape", "blackberry", and "watermelon". The cursor is positioned at the end of "watermelon". The status bar at the bottom of the terminal window displays the path "/grocery/shoppingList.txt[+]", the operating system as "[unix]", the time as "(15:22 26/08/2017)", and the file size as "7,11 All". Below the status bar, the text "-- INSERT --" is visible.

Merging branches

- After saving the file, add it to the staging area and then commit:

```
[3] ~/grocery (master|MERGING)
$ git add shoppingList.txt
```

```
[4] ~/grocery (master|MERGING)
$ git commit -m "Merged melons branch into master"
[master e18a921] Merged melons branch into master
```

Merging branches

- Now take a look at the log:

```
[5] ~/grocery (master)
$ git log --oneline --graph --decorate --all
*   e18a921 (HEAD -> master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
| /
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Merging branches

- Suggestion: look at the merge commit with git cat-file -p:

```
[6] ~/grocery (master)
$ git cat-file -p HEAD
tree 2916dd995ee356351c9b49a5071051575c070e5f
parent 6409527a1f06d0bbe680d461666ef8b137ac7135
parent a8c62190fb1c54d1034db78a87562733a6e3629c
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503754221 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503754221 +0200

Merged melons branch into master
```

Merging branches

Fast forwarding

- A merge not always generates a new commit; to test this case, try to merge the melons branch into a berries one:

```
[7] ~/grocery (master)
$ git checkout berries
Switched to branch 'berries'

[8] ~/grocery (berries)
$ git merge melons
Updating ef6c382..a8c6219
Fast-forward
  shoppingList.txt | 1 +
  1 file changed, 1 insertion(+)

[9] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ 
| * a8c6219 (HEAD -> berries, melons) Add a watermelon
| * ef6c382 Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Merging branches

- Move back the berries branch where it was using git reset:

```
[10] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry

[11] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
| /
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Merging branches

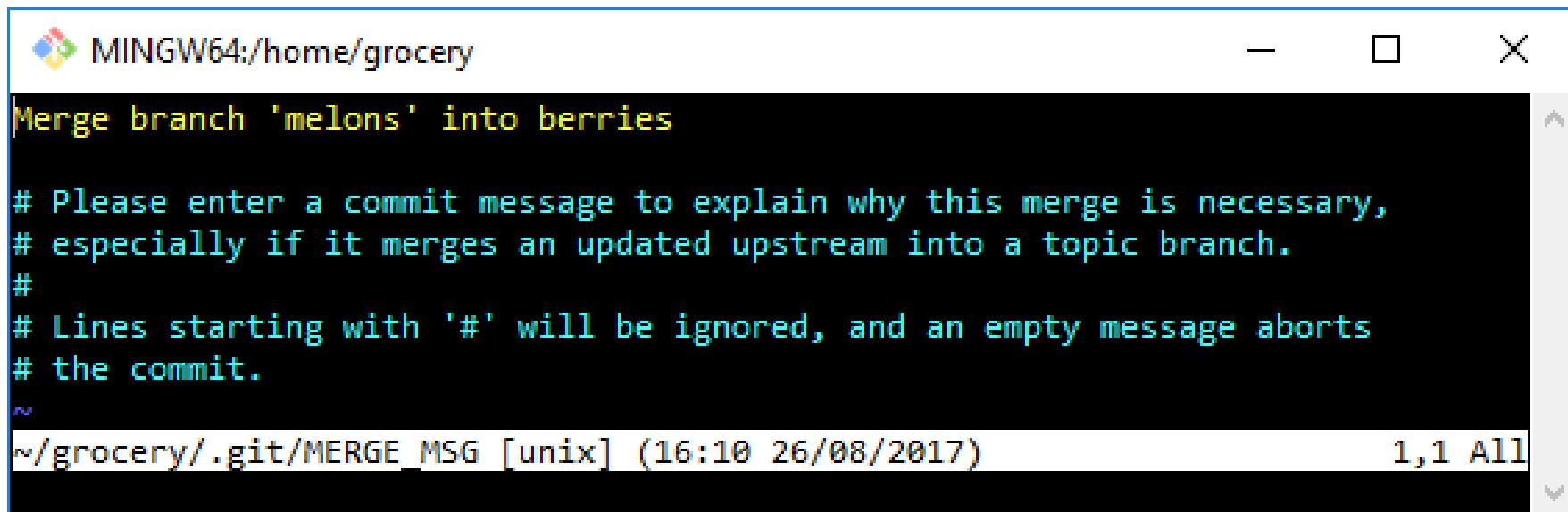
- We have just undone a merge, did you realize it?
- Okay, now do the merge again with the --no-ff option:

```
[12] ~/grocery (berries)
$ git merge --no-ff melons
```



Merging branches

- Git will now open your default editor to allow you to specify a commit message, as shown in the following screenshot:



The screenshot shows a terminal window titled "MINGW64:/home/grocery". The window contains the following text:

```
Merge branch 'melons' into berries

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~

~/grocery/.git/MERGE_MSG [unix] (16:10 26/08/2017) 1,1 All
```

Merging branches

- Accept the default message, save and exit:

```
[13] ~/grocery (berries)
```

```
Merge made by the 'recursive' strategy.--all  
shoppingList.txt | 1 +  
1 file changed, 1 insertion(+)
```

Merging branches

- Now a git log:

```
[14] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   cb912b2 (HEAD -> berries) Merge branch 'melons' into berries
|\ \
| | *   e18a921 (master) Merged melons branch into master
| | |
| | /|
| | /|
| * | a8c6219 (melons) Add a watermelon
| // 
* | ef6c382 Add a blackberry
| * 6409527 Add a grape
| * 603b9d1 Add a peach
| /
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Merging branches

- We are done with these experiments; anyway, I want to undo this merge, because I want to keep the repository as simple as possible to allow you to better understand the exercise we do together; go with a git reset --hard HEAD^:

```
[15] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry

[16] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Merging branches

Okay, now undo even the past merge we did on the master branch:

```
[17] ~/grocery (master)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[18] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| /
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Cherry picking

- Let's play with it a little bit.
- Assume you want to pick the blackberry from the berries branch, and then apply it into the master branch; this is the way:

```
[1] ~/grocery (master)
$ git cherry-pick ef6c382
error: could not apply ef6c382... Add a blackberry
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

Cherry picking

- Okay, the cherry pick raised a conflict, of course:

```
[2] ~/grocery (master|CHERRY-PICKING)
$ git diff
diff --cc shoppingList.txt
index 862debc,b05b25f..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,4 +1,9 @@@
banana
apple
orange
<<< HEAD
+peach
- grape
++grape
=====
+ blackberry
>>> ef6c382... Add a blackberry
```

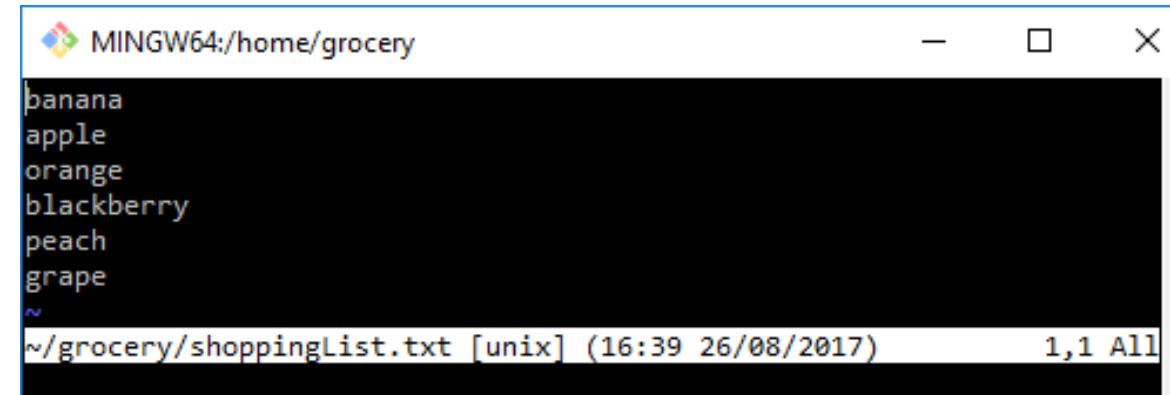
Cherry picking

- The fourth line of both the shoppingList.txt file versions has been modified with different fruits.
- Resolve the conflict and then add a commit:

[3] ~/grocery (master|CHERRY-PICKING)

\$ vi shoppingList.txt

- The following is a screenshot of my Vim console, and the files are arranged as I like:



A screenshot of a terminal window titled "MINGW64:/home/grocery". The window contains a list of fruit names: banana, apple, orange, blackberry, peach, grape. The cursor is at the bottom of the list, indicated by a tilde (~). The status bar at the bottom shows the path "~/grocery/shoppingList.txt [unix] (16:39 26/08/2017)" and the text "1,1 All".

Cherry picking

```
[4] ~/grocery (master|CHERRY-PICKING)
$ git add shoppingList.txt

[5] ~/grocery (master|CHERRY-PICKING)
$ git status
On branch master
You are currently cherry-picking commit ef6c382.
(all conflicts fixed: run "git cherry-pick --continue")
(use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes to be committed:

modified:   shoppingList.txt
```

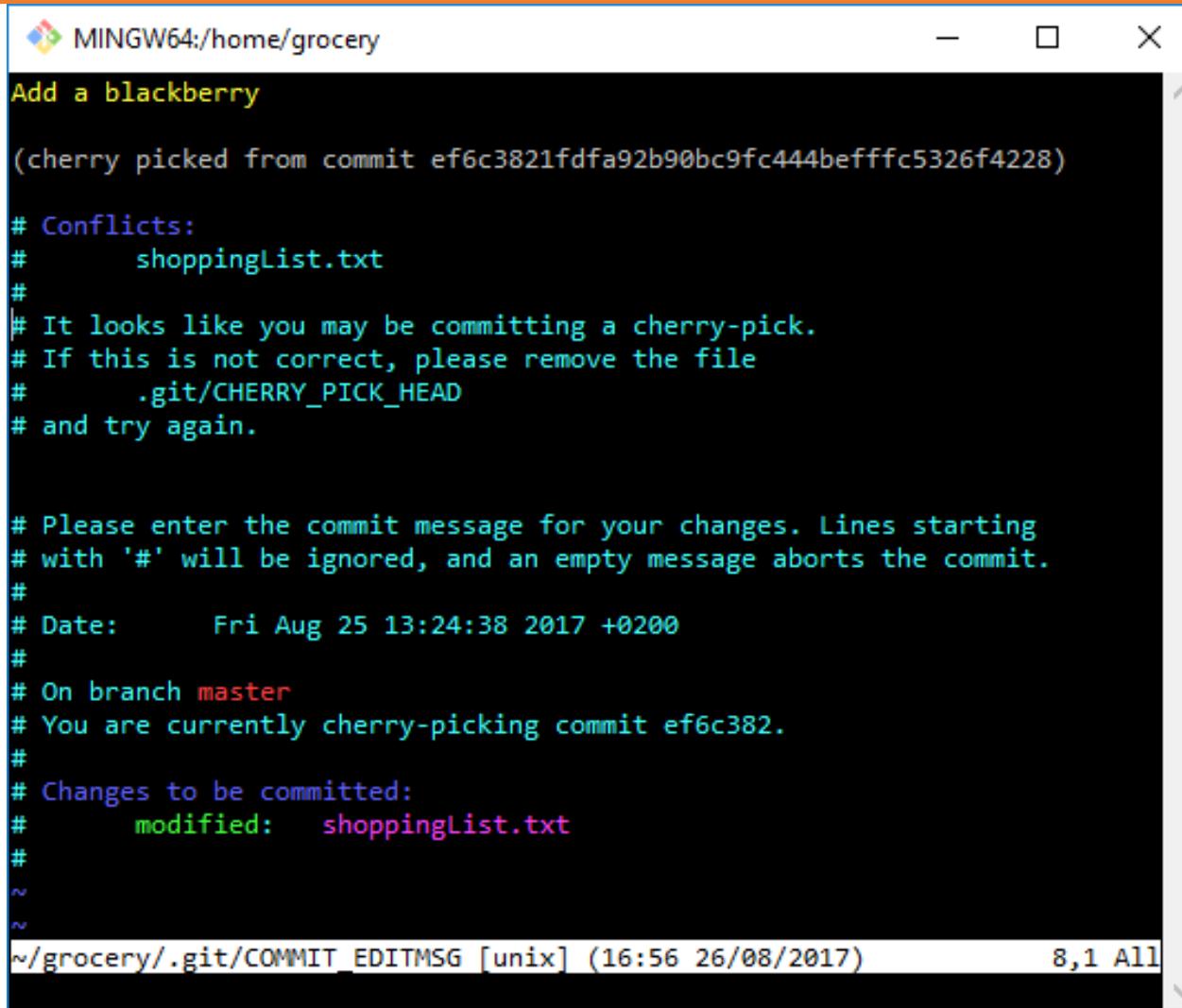
Cherry picking

- Now go on and commit:

```
[6] ~/grocery (master)
$ git commit -m "Add a cherry-picked blackberry"
On branch master
nothing to commit, working tree clean

[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 99dd471 (HEAD -> master) Add a cherry-picked blackberry
* 6409527 Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Cherry picking



MINGW64:/home/grocery

```
Add a blackberry

(cherry picked from commit ef6c3821fd9a92b90bc9fc444befffc5326f4228)

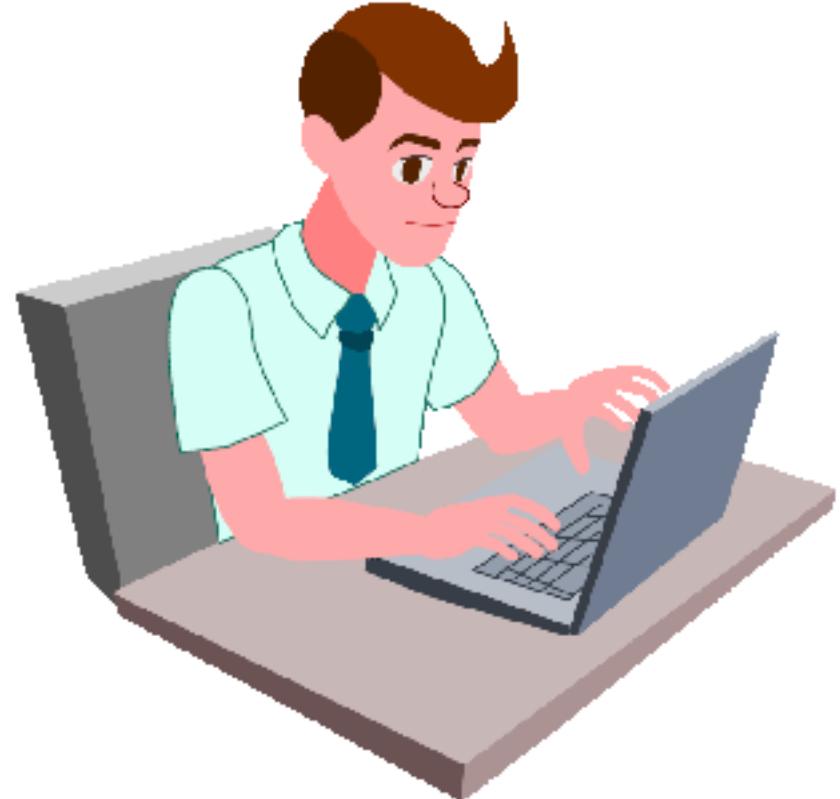
# Conflicts:
#       shoppingList.txt
#
# It looks like you may be committing a cherry-pick.
# If this is not correct, please remove the file
#       .git/CHERRY_PICK_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Aug 25 13:24:38 2017 +0200
#
# On branch master
# You are currently cherry-picking commit ef6c382.
#
# Changes to be committed:
#       modified:   shoppingList.txt
#
~
```

~/grocery/.git/COMMIT_EDITMSG [unix] (16:56 26/08/2017) 8,1 All

Summary

- This has been a very long lesson, I know.
- But now I think you know all you need to work proficiently with Git, at least in your own local repository.
- You know about working tree, staging area, and HEAD commit; you know about references as branches and HEAD.
- You know how to merge rebase, and cherry pick; and finally, you know how Git works under the hood, and this will help you from here on out.



"Complete Lab 6"

7. Resolving Merge Conflicts

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or software development environment.

Local Merge Conflicts

- Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file.
- To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.

Local Merge Conflicts

- Open Git Bash.
- Navigate into the local Git repository that has the merge conflict.

```
cd REPOSITORY-NAME
```

Local Merge Conflicts

```
$ git status
> # On branch branch-b
> # You have unmerged paths.
> #   (fix conflicts and run "git commit")
> #
> # Unmerged paths:
> #   (use "git add ..." to mark resolution)
> #
> # both modified:    styleguide.md
> #
> no changes added to commit (use "git add" and/or "git commit -a")
```

Local Merge Conflicts

- In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or branch-a.
- If you have questions, please

<<<<< HEAD

open an issue

=====

ask your question in IRC.

>>>>> branch-a

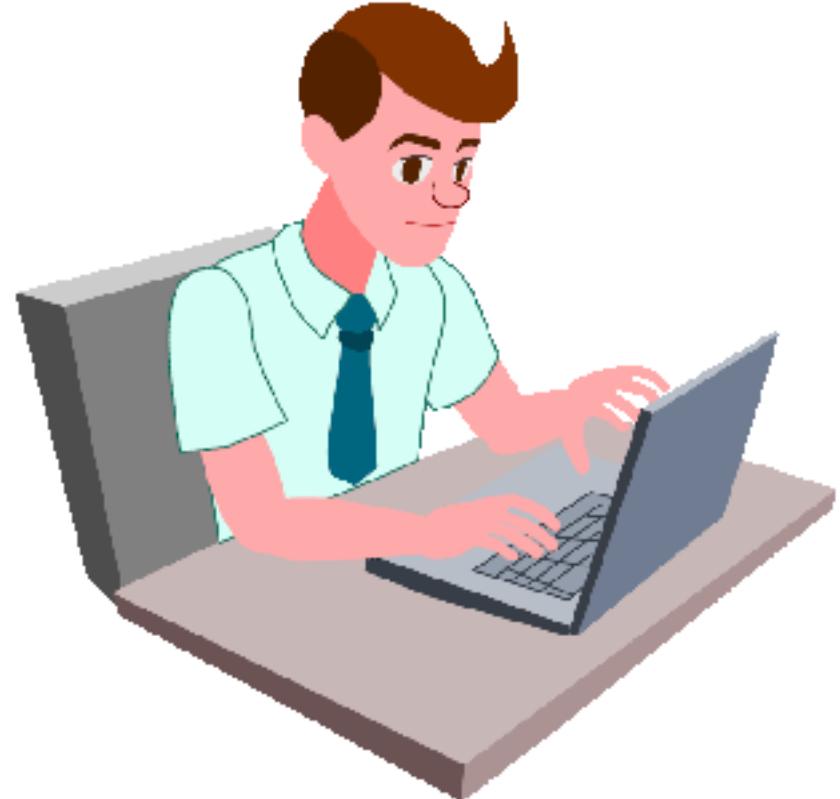
Local Merge Conflicts

- Add or stage your changes.

```
$ git add .
```

- Commit your changes with a comment.

```
$ git commit -m "Resolved merge conflict by incorporating  
both suggestions."
```



"Complete Lab 7"

8. Remote Repositories



Git Fundamentals - Working Remotely

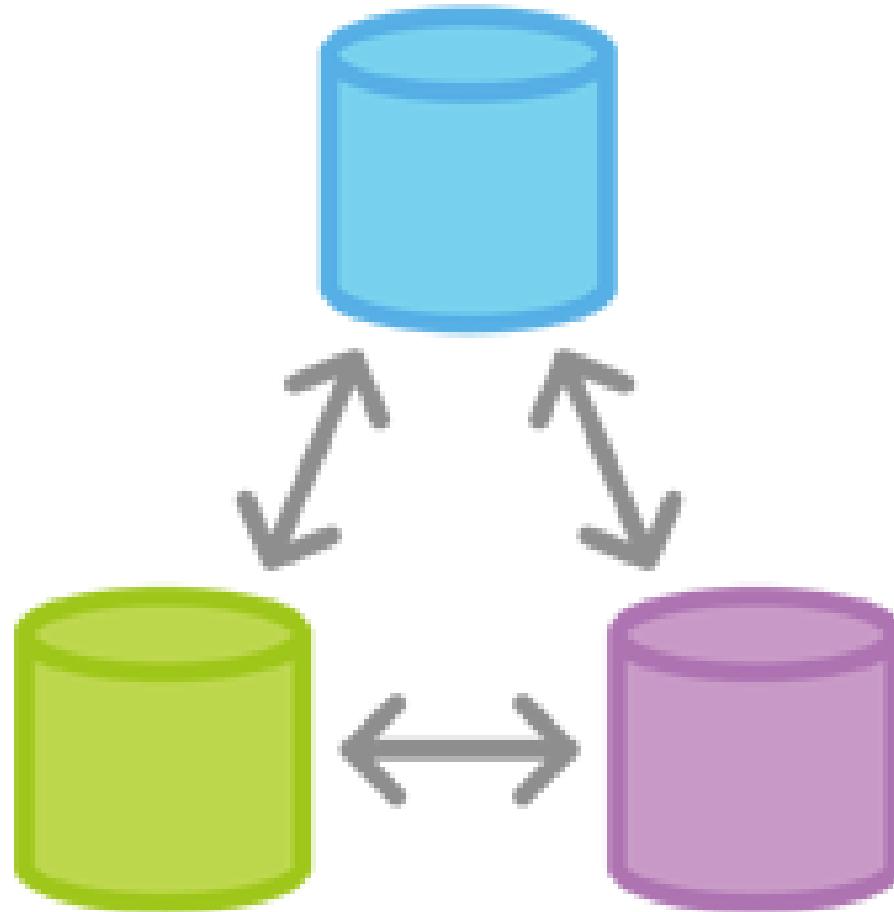
In this lesson, we finally start to work in a distributed manner, using remote servers as a contact point for different developers. These are the main topics we will focus on:

- Dealing with remotes
- Cloning a remote repository
- Working with online hosting services, such as GitHub

Working with remotes

- Git is a tool for versioning files, as you know, but it has been built with collaboration in mind.
- In 2005, Linus Torvalds had the need for a light and efficient tool to share the Linux kernel code, allowing him and hundreds of other people to work on it without going crazy.

Working with remotes



Working with remotes

Clone a local repository

- Create a new folder on your disk to clone our grocery repository:

```
[1] ~  
$ mkdir grocery-cloned
```

- Then clone the grocery repository using the git clone command:

```
[2] ~ $ cd grocery-cloned [3] ~/grocery-cloned $ git clone ~/grocery . Cloning into '.'....
```

Working with remotes

- Now, go directly to the point with a git log command:

```
[4] ~/grocery-cloned (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master, origin/master, origin/HEAD) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Working with remotes

- But don't worry: a local branch in which to work locally can be created by simply checking it out:

```
[5] ~/grocery-cloned (master)
```

```
$ git checkout berries
```

```
Branch berries set up to track remote branch berries from origin.  
Switched to a new branch 'berries'
```

Working with remotes

- Now, look at the log again:

```
[6] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* 6409527 (origin/master, origin/HEAD, master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (HEAD -> berries, origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Working with remotes

- Let's try:

```
[7] ~/grocery-cloned (berries)
$ echo "blueberry" >> shoppingList.txt
```

```
[8] ~/grocery-cloned (berries)
$ git commit -am "Add a blueberry"
[berries ab9f231] Add a blueberry
Committer: Santacroce Ferdinando <san@intre.it>
```

Working with remotes

- You can suppress this message by setting them explicitly:
`git config --global user.name "Your Name"`
`git config --global user.email you@example.com`
- After doing this, you may fix the identity used for this commit with the following code:

```
git commit --amend --reset-author
```

1 file changed, 1 insertion(+)

Working with remotes

OK, let's see what happened:

```
[9] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* ab9f231 (HEAD -> berries) Add a blueberry
| * 6409527 (origin/master, origin/HEAD, master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (origin/melons) Add a watermelon
| |
|/
|/
* | ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Working with remotes

- Now, we will try to push the modifications in the berries branch to the origin; the command is git push, followed by the name of the remote and the target branch:

```
[10] ~/grocery-cloned (berries)
$ git push origin berries
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 323 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
    ef6c382..ab9f231  berries -> berries
```

Working with remotes

- Now, we obviously want to see if, in the remote repository, there is a new commit in the berries branch; so, open the grocery folder in a new console and do git log:

```
[11] ~
$ cd grocery

[12] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* ab9f231 (berries) Add a blueberry
| * 6409527 (HEAD -> master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (melons) Add a watermelon
| |
|/
|/
* | ef6c382 Add a blackberry
|
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Working with remotes

Getting remote commits with git pull

- Now, it's time to experiment the inverse: retrieving updates from the remote repository and applying them to our local copy.
- So, make a new commit in the grocery repository, and then, we will download it into the grocery-cloned one:

```
[13] ~/grocery (master)
$ printf "\r\n" >> shoppingList.txt
```

Working with remotes

- I firstly need to create a new line, because due to the previous grape rebase, we ended having the shoppinList.txt file with no new line at the end, as echo "" >> <file> usually does:

```
[14] ~/grocery (master)
$ echo "apricot" >> shoppingList.txt
```

```
[15] ~/grocery (master)
$ git commit -am "Add an apricot"
[master 741ed56] Add an apricot
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Working with remotes

- Let's try git pull for now, then we will try to use git fetch and git merge separately.
- Go back to the grocery-cloned repository, switch to the master branch, and do a git pull:

```
[16] ~/grocery-cloned (berries)
$ git checkout master
Your branch is up-to-date with 'origin/master'.
Switched to branch 'master'
```

Working with remotes

- For now, go with git pull: the command wants you to specify the name of the remote you want to pull from, which is origin in this case, and then the branch you want to merge into your local one, which is master, of course:

```
[17] ~/grocery-cloned (master)
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
 * branch      master    -> FETCH_HEAD
   6409527..741ed56  master    -> origin/master
Updating 6409527..741ed56
Fast-forward
  shoppingList.txt | 3 +++
  1 file changed, 2 insertions(+), 1 deletion(-)
```

Working with remotes

- OK, now I want you to try doing these steps in a separate manner; create the umpteenth new commit in the grocery repository, the master branch:

```
[18] ~/grocery (master)
$ echo "plum" >> shoppingList.txt

[19] ~/grocery (master)
$ git commit -am "Add a plum"
[master 50851d2] Add a plum
1 file changed, 1 insertion(+)
```

Working with remotes

- Now perform a git fetch on grocery-cloned repository:

```
[20] ~/grocery-cloned (master)
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
  741ed56..50851d2  master      -> origin/master
```

Working with remotes

- Do a git status now:

```
[21] ~/grocery-cloned (master)
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working tree clean
```

Working with remotes

- Now, let's sync with a git merge; to merge a remote branch, we have to specify, other than the branch name, even the remote one, as we did in the git pull command previously:

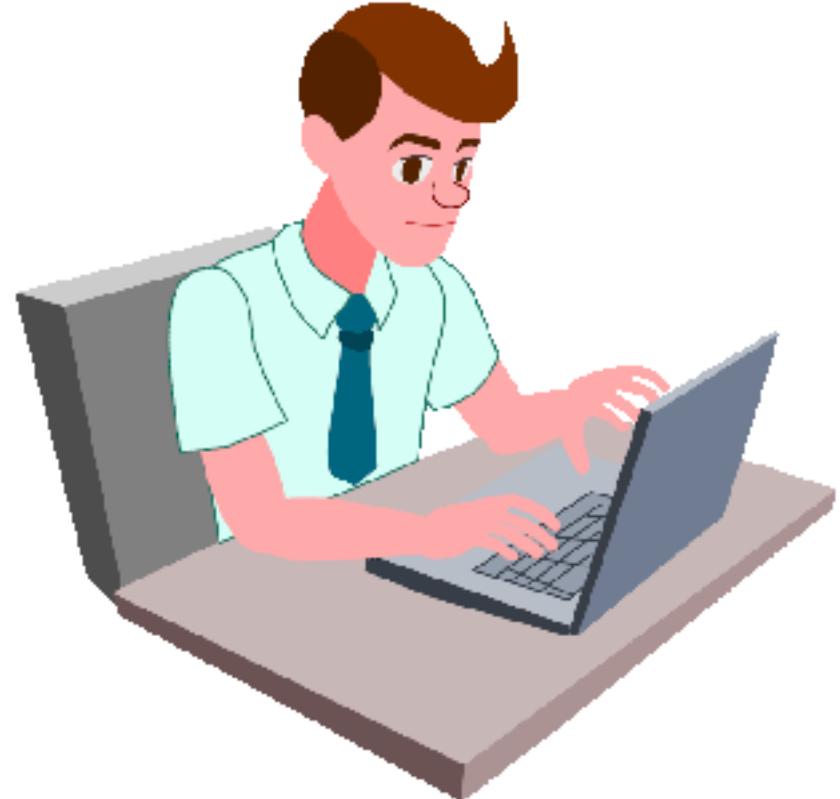
```
[22] ~/grocery-cloned (master)
$ git merge origin master
Updating 741ed56..50851d2
Fast-forward
  shoppingList.txt | 1 +
  1 file changed, 1 insertion(+)
```

Working with remotes

How Git keeps track of remotes

- Git stores remote branch labels in a similar way to how it stores the local branches ones; it uses a subfolder in refs for the scope, with the symbolic name we used for the remote, in this case origin, the default one:

```
[23] ~/grocery-cloned (master)
$ ll .git/refs/remotes/origin/
total 3
drwxr-xr-x 1 san 1049089 0 Aug 27 11:25 .
drwxr-xr-x 1 san 1049089 0 Aug 26 18:19 ../
-rw-r--r-- 1 san 1049089 41 Aug 26 18:56 berries
-rw-r--r-- 1 san 1049089 32 Aug 26 18:19 HEAD
-rw-r--r-- 1 san 1049089 41 Aug 27 11:25 master
```



"Complete Lab 8"

9. Reviewing the Commit History



Using Git Log

- In modern Git, you can trace the evolution of the line range within the file using git log -L, which is currently limited to walk starting from a single revision (zero or one positive revision arguments) and a single file.
- The range is given either by denoting the start and end of the range with -L <start>,<end>:<file> (where either <start> or <end> can be the line number or /regexp/), or a function to track with -L :<funcname regexp>:<file>.

Selecting and formatting the git log output

- Now that you know how to select revisions to examine and to limit which revisions are shown (selecting those that are interesting).
- There is a huge number and variety of options of the git log command available for this.

Predefined and user defined output formats

```
$ git log --pretty="%h - %an, %ar : %s"  
50f84e3 - Junio C Hamano, 7 days ago : Update draft release notes  
0953113 - Junio C Hamano, 10 days ago : Second batch for 2.1  
afa53fe - Nick Alcock, 2 weeks ago : t5538: move http push tests out
```

Using Git Log

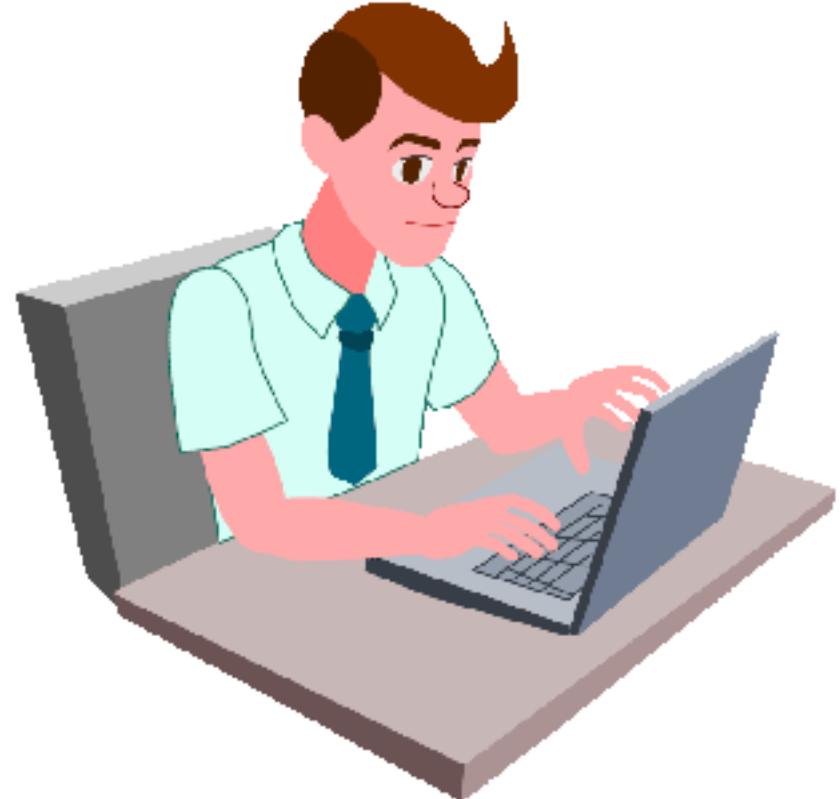


Placeholder	Description of output
%H	Commit hash (full SHA-1 identifier of revision)
%h	Abbreviated commit hash
%an	Author name
%ae	Author e-mail
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cr	Committer date, relative
%s	Subject (first line of a commit message, describing revision)
%%	A raw %

Using Git Log

```
$ git log --graph --decorate --oneline origin/maint
* bce14aa (origin/maint) Sync with 1.9.4
|\ 
| * 34d5217 (tag: v1.9.4) Git 1.9.4
| * 12188a8 Merge branch 'rh/prompt' into maint
| |\ 
| * \| 64d8c31 Merge branch 'mw/symlinks' into maint
| |\ \
* ||| d717282 t5537: re-drop http tests
* ||| e156455 (tag: v2.0.0) Git 2.0
```





"Complete Lab 9"

10. Improving Your Daily Workflow



Git aliases

Shortcuts to common commands

- One thing you may find useful is to shorten common commands such as git checkout and so on; therefore, useful aliases can include the following:

[1] ~/grocery (master)

```
$ git config --global alias.co checkout
```

[2] ~/grocery (master)

```
$ git config --global alias.br branch
```

[3] ~/grocery (master)

```
$ git config --global alias.ci commit
```

[4] ~/grocery (master)

```
$ git config --global alias.st status
```

Git aliases

- Another common practice is to shorten a command, adding one or more options you use all the time; for example, set a git cm <commit message> command shortcut to the alias git commit -m <commit message>:

[5] ~/grocery (master)

```
$ git config --global alias.cm "commit -m"
```

[6] ~/grocery (master)

```
$ git cm "My commit message"
```

On branch master

nothing to commit, working tree clean



Git aliases

- The classic example is the git unstage alias:

[1] ~/grocery (master)

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

- With this alias, you can remove a file from the index in a more meaningful way, compared to the equivalent git reset HEAD -- <file> syntax:

[2] ~/grocery (master)

```
$ git unstage myFile.txt
```

Git aliases

- Now behaves the same as:

[3] ~/grocery (master)

```
$ git reset HEAD -- myFile.txt
```



git undo

- Want a fast way to revert the last ongoing commit? Create a git undo alias:

[1] ~/grocery (master)

```
$ git config --global alias.undo 'reset --soft HEAD~1'
```

Git aliases

git last

- A git last alias is useful to read about your last commit:

```
[1] ~/grocery (master)
```

```
$ git config --global alias.last 'log -1 HEAD'
```

```
[2] ~/grocery (master)
```

```
$ git last
```

```
commit b25ffa60f44f6fc50e81181cab87ed3dbf3b172c
```

```
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
```

```
Date: Thu Jul 27 15:12:48 2017 +0200
```

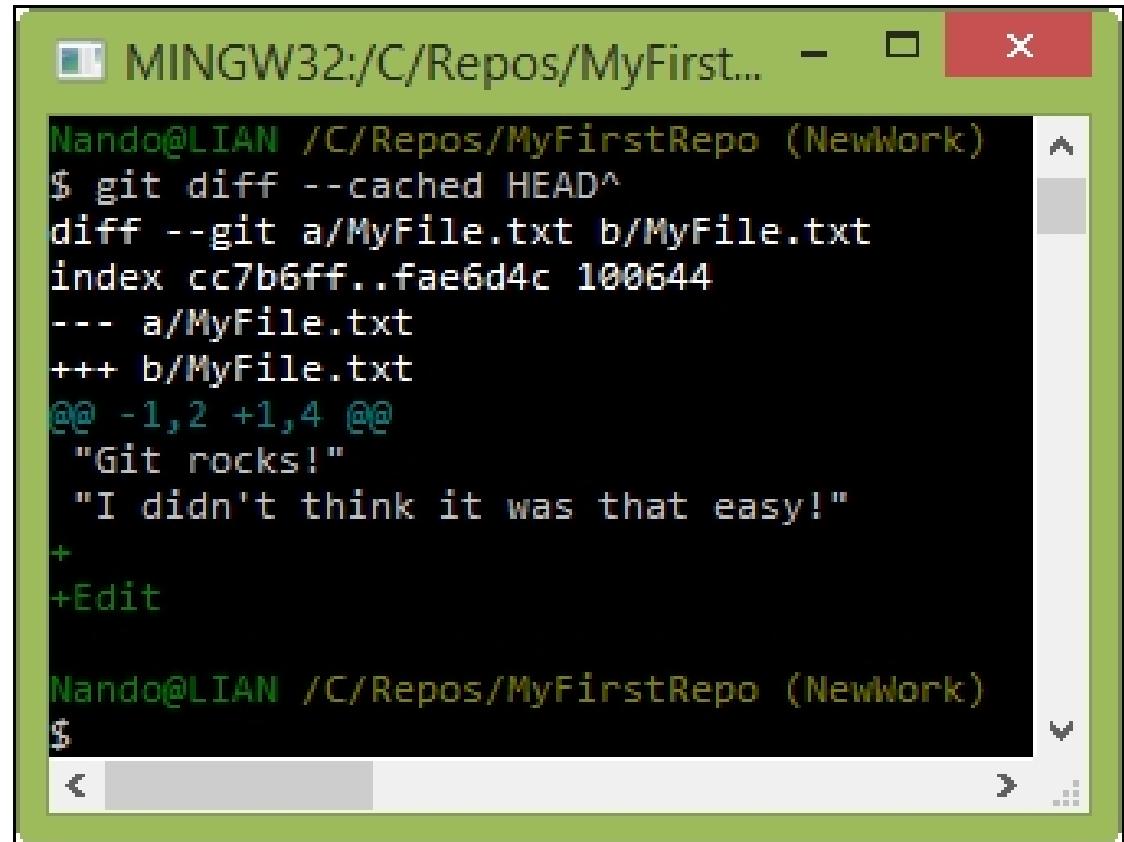
Add an apricot

Git aliases

git difflast

- With the git difflast alias, you can see a diff against your last commit:

```
$ git config --global alias.difflast  
'diff --cached HEAD^'
```



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirst...". The command entered is \$ git diff --cached HEAD^, which outputs a diff between the current state and the previous commit. The diff shows changes to a file named MyFile.txt, where the line "Git rocks!" has been added and the line "I didn't think it was that easy!" has been removed.

```
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git diff --cached HEAD^
diff --git a/MyFile.txt b/MyFile.txt
index cc7b6ff..fae6d4c 100644
--- a/MyFile.txt
+++ b/MyFile.txt
@@ -1,2 +1,4 @@
 "Git rocks!"
 "I didn't think it was that easy!"
+
+Edit

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$
```

Git aliases

Advanced aliases with external commands

- If you want the alias to run external shell commands, instead of a Git sub-command, you have to prefix the alias with a !:

```
$ git config --global alias.echo !echo
```

- Suppose you are annoyed by the canonical git add <file> plus git commit <file> sequence of commands, and you want to do it in a single shot; you can call the git command twice in sequence by creating this alias:

```
$ git config --global alias.cm '!git add -A && git commit -m'
```

Removing an alias

- Removing an alias is quite easy; you have to use the `--unset` option, specifying the alias to remove.
- For example, if you want to remove the `cm` alias, you have to run:

```
$ git config --global --unset alias.cm
```

Aliasing the git command itself

- I've already said I'm a bad typist; if you are too, you can alias the git command itself (using the default alias command in Bash):

```
$ alias gti='git'
```

Useful techniques

- Append a new fruit to the shopping list, then try to switch branch.
- Git won't allow you to do so, because with the checkout you would lose your local (not yet committed) changes to the shoppingList.txt file.
- So, type the git stash command; your changes will be set apart and removed from your current branch, letting you switch to another one (berries, in this case).

Useful techniques

Git commit amend - modify the last commit

- This trick is for people that don't double-check what they're doing.
- If you have pressed the enter key too early, there's a way to modify the last commit message or add that file you forgot, using the git commit command with the --amend option:

```
$ git commit --amend -m "New commit message"
```

Useful techniques

- The last tip I want to suggest is to use the Git GUI:
[3] ~/Spoon-Knife (master)
\$ git gui blame README.md

The screenshot shows the Git Gui (Spoon-Knife) interface. The title bar reads "Git Gui (Spoon-Knife): File Viewer". The menu bar includes "Repository", "Edit", and "Help". The status bar at the bottom says "Annotation complete.". The main area displays the content of the README.md file. The first few lines are annotated with commit details:

```
bb4c bb4c 1 ### Well hello there!
TO TO 2
| |
| 3 This repository is meant to provide an example for *forking* a repository on GitHub.
| 4
| 5 Creating a *fork* is producing a personal copy of someone else's project. Forks are
| 6
| 7 After forking this repository, you can make some changes to the project, and submit
| 8
d0dd d0dd 9 For some more information on how to fork a repository, [check out our guide, "For
```

Below the file content, a commit summary is shown:

```
commit bb4cc8d3b2e14b3af5df699876dd4ff3acd00b7f
Author: The Octocat <octocat@nowhere.com> Tue Feb 4 23:38:36 2014
Committer: The Octocat <octocat@nowhere.com> Thu Feb 13 00:18:55 2014
```

At the bottom, a note says "Create styles.css and updated README".

Tricks

- If you want to set up a bare repository, you only have to use the --bare option:

```
$ git init --bare NewRepository.git
```

- As you may have noticed, I called it NewRepository.git, using a .git extension; this is not mandatory, but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with a .git extension.

Converting a regular repository to a bare one

- It can happen that you start working on a project in a local repository, and then you feel the need to move it to a centralized server to make it available for other people or from other locations.
- You can easily convert a regular repository to a bare one using the `git clone` command with the same `--bare` option:

```
$ git clone --bare my_project my_project.git
```

Archiving the repository

- To archive the repository without including versioning information, you can use the git archive command; there are many output formats but the classic one is the .zip one:

```
$ git archive master --format=zip --output=../repoBackup.zip
```

Tricks

- Please note that using this command is not the same as backing up folders in a filesystem.
- As you will have noticed, the git archive command can produce archives in a smarter way, including only files in a branch or even in a single commit.
- For example, by doing this you are archiving only the last commit:

```
$ git archive HEAD --format=zip --output=../headBackup.zip
```

Bundling the repository

- Another interesting command is the git bundle command.
- With git bundle, you can export a snapshot from your repository and then restore it wherever you want.
- Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you can create a `repo.bundle` file of the master branch:

```
$ git bundle create ../repo.bundle master
```

Tricks

- With this other command, we can restore the bundle in the other computer using the git clone command:

```
$ cd /OtherComputer/Folder  
$ git clone repo.bundle repo -b master
```

Summary

- In this lesson, we enhanced our knowledge about Git and its wide set of commands.
- We discovered how configuration levels work, and how to set our preferences using Git by, for example, adding useful command aliases to the shell.
- Then we looked at how Git deals with stashes, providing the way to shelve then and reapply changes.

Aliases

- An alias is a nice way to configure long and/or complicated Git commands to represent short useful ones.
- An alias is simply a configuration entry under the alias section, It is usually configured to --global to apply it everywhere.

Getting ready

- In this example, we will use the jgit repository, Navigating Git, with the master branch pointing at b14a93971837610156e815ae2eee3baaa5b7a44b. Navigating Git, or clone the repository again, as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit  
$ git checkout master && git reset --hard b14a939
```

How to do it...

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

- Now, try to run git st in the jgit repository as follows:

```
$ git st  
# On branch master  
nothing to commit, working directory clean
```

How to do it...

- The alias method is also good to create the Git commands you think are missing in Git.
- One of the common Git aliases is unstage, which is used to move a file out of the staging area, as shown in the following command:

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

How to do it...

- Try to edit the README.md file in the root of the jgit repository and add it in the root.
- Now, git status/git st should display something like the following:

```
$ git st
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: README.md
```

How to do it...

```
$ git unstage README.md
Unstaged changes after reset:
M      README.md

$ git st
# On branch master
# 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:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```

How to do it...

- Let's say you want the number of lines added and deleted for each file in the commit displayed along with some common commit data.
- For this, we can create the following alias so we don't have to type everything each time:

```
$ git config --global alias.ll "log --  
pretty=format:\"%C(yellow)%h%Cred%d %Creset%s  
%Cgreen(%cr) %C(bold blue)<%an>%Creset\" --numstat"
```

How to do it...

```
$ git ll
b14a939 (HEAD, master) Prepare 3.3.0-SNAPSHOT builds (8 days ago) <Matthias Sohn>
6      6      org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.ant.test/pom.xml
3      3      org.eclipse.jgit.ant/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.ant/pom.xml
4      4      org.eclipse.jgit.archive/META-INF/MANIFEST.MF
2      2      org.eclipse.jgit.archive/META-INF/SOURCE-MANIFEST.MF
1      1      org.eclipse.jgit.archive/pom.xml
6      6      org.eclipse.jgit.console/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.console/pom.xml
12     12     org.eclipse.jgit.http.server/META-INF/MANIFEST.MF
...
...
```

How to do it...

- The examples can be used when resolving conflicts from a rebase or merge.
- In your `~/.gitconfig` file under [alias], add the following:

```
editconflicted = "!f() {git ls-files --unmerged | cut -f2 | sort -u ;  
}; $EDITOR 'f'"
```

How to do it...

- In the jgit repository, we can create two branches at an earlier point in time and merge these two branches:

```
$ git branch A 03f78fc
```

```
$ git branch B 9891497
```

```
$ git checkout A
```

```
Switched to branch 'A'
```

```
$ git merge B
```

How to do it...

- Now that we have solved all the merge conflicts, it is time to add all of those files before we conclude the merge.
- Luckily, we can create an alias that can help us with that, as follows:

```
addconflicted = "!f() { git ls-files --unmerged | cut -f2 | sort -u ; }; git add 'f'"
```

How to do it...

- Now, we can run git addconflicted.
- Later, git status will tell us that all the conflicted files are added:

```
$ git st
On branch A
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

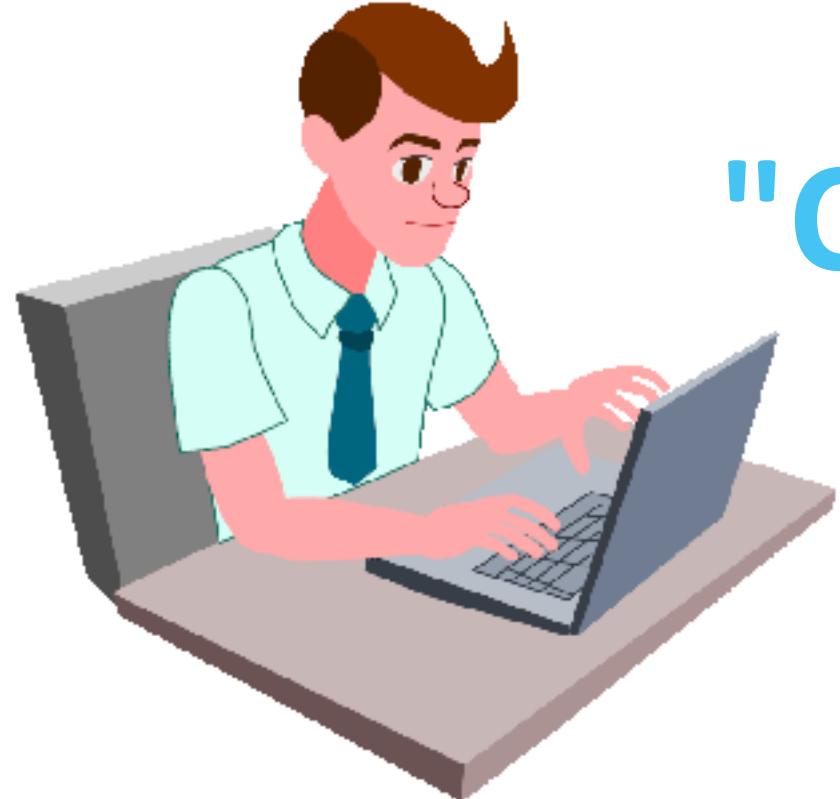
Changes to be committed:

modified: org.eclipse.jgit.console/META-INF/MANIFEST.MF
modified: org.eclipse.jgit.console/pom.xml
modified: org.eclipse.jgit.http.server/META-INF/MANIFEST.MF
modified: org.eclipse.jgit.http.server/pom.xml
modified: org.eclipse.jgit.http.test/META-INF/MANIFEST.MF
modified: org.eclipse.jgit.http.test/pom.xml
...
...
```

How to do it...

- Now we can conclude the merge with git commit:

```
$ git commit  
[A 94344ae] Merge branch 'B' into A
```



"Complete Lab 10 & 11

11. Continuous Integration / Continuous Delivery (CI/CD)



CI/CD explained

- CI/CD falls under DevOps (the joining of development and operations teams) and combines the practices of continuous integration and continuous delivery.
- CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production, encompassing the build, test (including integration tests, unit tests, and regression tests), and deploy phases, as well as infrastructure provisioning.

What is continuous integration (CI)?

- Continuous integration is the practice of integrating all your code changes into the main branch of a shared source code repository early and often, automatically testing each change when you commit or merge them, and automatically kicking off a build.
- With continuous integration, errors and security issues can be identified and fixed more easily, and much earlier in the development process.
- By merging changes frequently and triggering automatic testing and validation processes, you minimize the possibility of code conflict, even with multiple developers working on the same application..

What is continuous delivery (CD)?

- Continuous delivery is a software development practice that works in conjunction with CI to automate the infrastructure provisioning and application release process.
- Once code has been tested and built as part of the CI process, CD takes over during the final stages to ensure it's packaged with everything it needs to deploy to any environment at any time.
- CD can cover everything from provisioning the infrastructure to deploying the application to the testing or production environment.

What is continuous deployment?

- Continuous deployment enables organizations to deploy their applications automatically, eliminating the need for human intervention.
- With continuous deployment, DevOps teams set the criteria for code releases ahead of time and when those criteria are met and validated, the code is deployed into the production environment.
- This allows organizations to be more nimble and get new features into the hands of users faster.

What is continuous testing?

In continuous testing, various types of tests are performed within the CI/CD pipeline. These can include:

- Unit testing, which checks that individual units of code work as expected
- Integration testing, which verifies how different modules or services within an application work together
- Regression testing, which is performed after a bug is fixed to ensure that specific bug won't occur again

There are eight fundamental elements of CI/CD that help ensure maximum efficiency for your development lifecycle. They span development and deployment. Include these fundamentals in your pipeline to improve your DevOps workflow and software delivery:

- A single source repository
- Frequent check-ins to main branch
- Automated builds
- Self-testing builds
- Frequent iterations
- Stable testing environments
- Maximum visibility
- Predictable deployments anytime

CI/CD fundamentals

The benefits of CI/CD implementation

Companies and organizations that adopt CI/CD tend to notice a lot of positive changes. Here are some of the benefits you can look forward to as you implement CI/CD:

- Happier users and customers
- Accelerated time-to-value
- Less fire fighting
- Free up developers' time
- Less context switching
- Reduce burnout
- Recover faster

Why GitLab CI/CD?

- In order to complete all the required fundamentals of full CI/CD, many CI platforms rely on integrations with other tools to fulfill those needs.
- Many organizations have to maintain costly and complicated toolchains in order to have full CI/CD capabilities.
- This often means maintaining a separate SCM like Bitbucket or GitHub, and connecting to a separate testing tool that connects to their CI tool, that connects to a deployment tool like Chef or Puppet, that also connects to various security and monitoring tools.

GitLab CI

- GitLab CI is a feature that helps perform the Continuous Integration (CI) of software components.
- When several developers work together using a versioning system, problems can arise when changes made by one developer break the product as a whole.
- The best way to make sure this happens less often, or at least early in the process, is to use integration tests more frequently, hence the name continuous.

GitLab CI

- Forrester classified GitLab as a leader in CI in The Forrester Wave: Continuous Integration Tools, Q3 2017.
- This is shown in the following diagram:



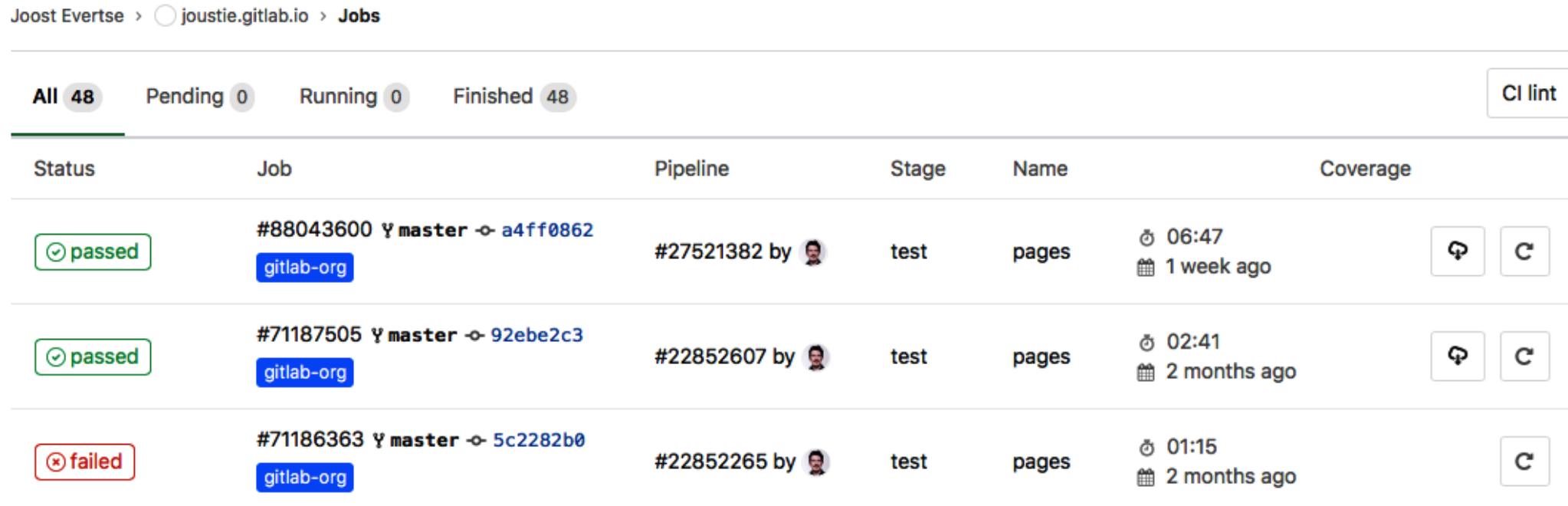
Pipelines and jobs

Joost Evertse > [joustie.gitlab.io](#) > Pipelines

All 47	Pending 0	Running 0	Finished 47	Branches	Tags	Run Pipeline	Clear Runner Caches	CI Lint
Status	Pipeline		Commit	Stages				
passed	#27521382 by 	latest	 master -o a4ff0862  Translated minecraft post	 	⌚ 00:06:47	 		
passed	#22852607 by 		 master -o 92ebe2c3  Update .gitlab-ci.yml	 	⌚ 00:02:41	 		
failed	#22852265 by 		 master -o 5c2282b0  Update .gitlab-ci.yml		⌚ 00:01:15	 		
failed	#22852093 by 		 master -o 9f2c8edd  Update .gitlab-ci.yml		⌚ 00:01:28	 		

Pipelines and jobs

- Alternatively, you can view all jobs, by going to the Pipelines' Jobs page, as shown in the following screenshot:

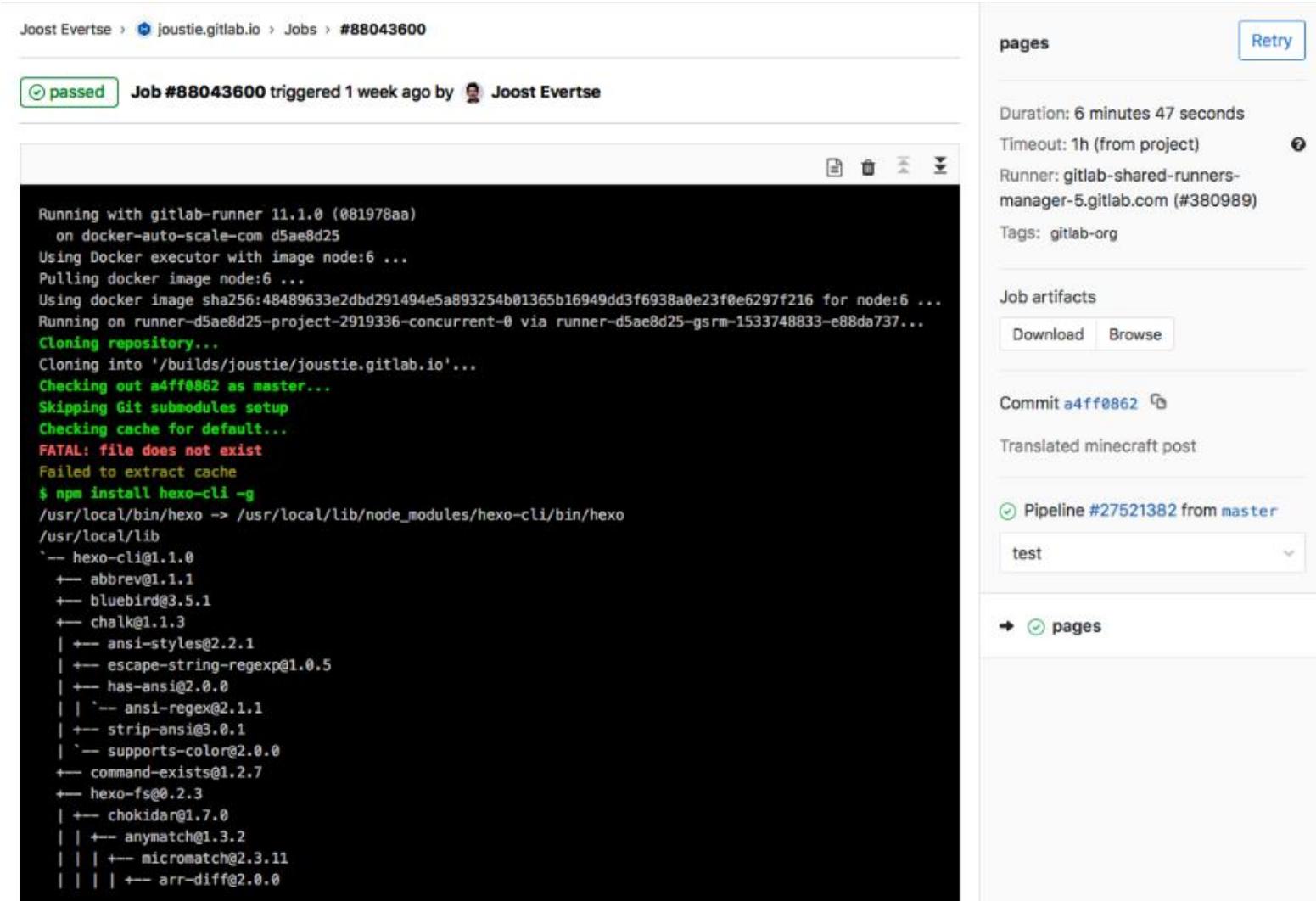


The screenshot shows a list of completed pipelines from the 'Jobs' page. The top navigation bar includes 'Joost Evertse > joustie.gitlab.io > Jobs'. Below the navigation, there are filters: 'All 48' (selected), 'Pending 0', 'Running 0', 'Finished 48', and a 'CI lint' button. The main table has columns: Status, Job, Pipeline, Stage, Name, and Coverage. Three rows are listed:

Status	Job	Pipeline	Stage	Name	Coverage
passed	#88043600 ⚡ master -> a4ff0862 gitlab-org	#27521382 by 🧑	test	pages	⌚ 06:47 📅 1 week ago
passed	#71187505 ⚡ master -> 92ebe2c3 gitlab-org	#22852607 by 🧑	test	pages	⌚ 02:41 📅 2 months ago
failed	#71186363 ⚡ master -> 5c2282b0 gitlab-org	#22852265 by 🧑	test	pages	⌚ 01:15 📅 2 months ago

Pipelines and jobs

- You can check the log of a job by clicking on the status of the job (for example, failed or passed). You can debug why some jobs fail and see exactly what happened:



The screenshot shows a GitLab pipeline job log for job #88043600, triggered 1 week ago by Joost Evertse. The job status is passed. The log output is as follows:

```
Running with gitlab-runner 11.1.0 (081978aa)
on docker-auto-scale-com d5ae8d25
Using Docker executor with image node:6 ...
Pulling docker image node:6 ...
Using docker image sha256:48489633e2dbd291494e5a893254b01365b16949dd3f6938a0e23f0e6297f216 for node:6 ...
Running on runner-d5ae8d25-project-2919336-concurrent-0 via runner-d5ae8d25-gsrm-1533748833-e88da737...
Cloning repository...
Cloning into '/builds/joustie/joustie.gitlab.io'...
Checking out a4ff0862 as master...
Skipping Git submodules setup
Checking cache for default...
FATAL: file does not exist
Failed to extract cache
$ npm install hexo-cli -g
/usr/local/bin/hexo => /usr/local/lib/node_modules/hexo-cli/bin/hexo
/usr/local/lib
`-- hexo-cli@1.1.0
  +-- abbrev@1.1.1
  +-- bluebird@3.5.1
  +-- chalk@1.1.3
    +-- ansi-styles@2.2.1
    +-- escape-string-regexp@1.0.5
    +-- has-ansi@2.0.0
    | +-- ansi-regex@2.1.1
    +-- strip-ansi@3.0.1
    +-- supports-color@2.0.0
    +-- command-exists@1.2.7
    +-- hexo-fs@0.2.3
    +-- chokidar@1.7.0
    | +-- anymatch@1.3.2
    | +-- micromatch@2.3.11
    | +-- arr-diff@2.0.0
```

The log indicates a fatal error: "FATAL: file does not exist" during the npm install process. The error occurs while trying to extract a cache file. The log also shows the dependency tree for the hexo-cli package.

Issues with the old runner

- The main problem with the old runner is that it could only run one concurrent job at a time.
- If you wanted to run more, you could either set up a new server or create an additional user to build jobs.
- Secondly, it always ran projects on the server shell, This made it really hard to test projects using different versions of Ruby or any other dependencies.

Switching to Go

- Go (or Golang) is a new language (less than 10 years old).
- It is already widely used by some impressive parties, such as Docker (<https://docker.com>), Google, Kubernetes (<https://kubernetes.io>), and Prometheus (<https://prometheus.io>).
- Go is a versatile tool that can help you to program at a low level, close to the operating system or at a high level in a language such as Java.
- It is perfectly suited to creating systems software.

Switching to Go

- The project can be found at <https://gitlab.com/gitlab-org/gitlab-runner>:



A screenshot of a GitLab project page for "gitlab-runner". The page features a dark blue header with the GitLab logo and navigation links like "Dashboard", "Search", "Help Center", and "Logout". Below the header, there's a banner with the text "gitlab-runner" and a small orange fox icon. The main content area shows the project details: "GitLab Runner" and "Project ID: 250833". Below this, there are social sharing icons for GitHub, LinkedIn, and others. At the bottom, there are links for "Files", "Commits", "Branches", "Tags", "Readme", "Changelog", "LICENSE", "Contribution guide", and "CI/CD configuration".

gitlab-runner

GitLab Runner

Project ID: 250833

Star 1000 Fork 778 SSH git@gitlab.com:gitlab-org/gitl

Files (130.6 MB) Commits (3,189) Branches (124) Tags (201) Readme Changelog LICENSE Contribution guide CI/CD configuration

Build, test, deploy, and monitor your code from a single application

- We believe a single application that offers visibility across the entire
- SDLC is the best way to ensure that every development stage is included
- and optimized. When everything is under one roof, it's as easy to pinpoint
- workflow bottlenecks and evaluate the impact each element has on
- deployment speed. GitLab has CI/CD built right in, no plugins required.

Breaking down .gitlab-ci.yml

- A basic .gitlab-ci.yml file might look something like this:

before_script:

```
- apt-get update -qq && DEBIAN_FRONTEND=noninteractive apt-get install -y -  
qq ca-certificates git php php-xml  
- php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"  
- php composer-setup.php  
- php composer.phar install
```

phpunit:

script:

```
- vendor/bin/phpunit tests/ROT13FormatterTest
```

The script parameter

- The most common parameter to a job, script is used to define any commands that should be run in the job. In our preceding example, we have the following:

phpunit:

script:

- vendor/bin/phpunit tests/ROT13FormatterTest

The stage parameter

- Stages are defined at the top level of the YAML file and are used to define separate blocks of jobs, which can be executed in parallel.
- They also define the order in which stages are run. In each job, the stage parameter can be used to define which build stage a job is in, thus grouping together similar jobs and allowing for jobs to depend on other jobs having finished.
- The following is an example of defining and using stages:

```
stages:  
  - build  
  - test  
  - deploy  
  
job 1:  
  stage: build  
  script: npm run build dependencies  
  
job 2:  
  stage: build  
  script: npm run build artifacts  
  
job 3:  
  stage: test  
  script: npm run test  
  
job 4:  
  stage: deploy  
  script: npm run deploy
```

The only and except parameters

- Please note that only and except don't have to be mutually exclusive either; you can use a combination of them to have more fine-grained control over when a job will be executed.
- An example of this is as follows:

job:

only:

- /`^iss-.*$/`

except:

- tags

The tags parameter

- You can specify a tag or tags in this parameter, which will limit this job to only being executed on Runners that also have the same tag.
- Please note that the tags are additive, so if you specify multiple tags, the job will only be executed on a Runner that has all of those tags present:

job:

tags:

- php

- postgres

The tags parameter

- One example use case of when is as follows:

stages:

- build
- cleanup_build

build_job:

stage: build

script:

- webpack

cleanup_build_job:

stage: cleanup_build

script:

- rm /dist/*

when: on_failure

cache – paths

- This is an array of paths to files and/or directories that should be cached.
- You can also use the asterisk wildcard character (*). This is demonstrated in the following code snippet:

build:

 cache:

 paths:

- binaries/*.apk
- .config

cache – key

- This takes a string that can be used to create separate caches for different jobs or branches, like so:

test:

cache:

key: "\$CI_COMMIT_REF_SLUG"

paths:

- binaries/

cache – key

- In this example, we use an inbuilt default variable provided by GitLab CI – `CI_COMMIT_REF_SLUG` – that is equal to the branch/tag name of the commit.
- In this case, GitLab CI will maintain a separate cache for each branch. Next up, let's look at jobs with different paths cached:

```
stages:  
  - build  
  - test  
  
build_job:  
  stage: build  
  script: npm run build  
  cache:  
    key: build-key  
    paths:  
      - public/  
  
test_job:  
  stage: test  
  script: npm run test  
  cache:  
    key: test-key  
    paths:  
      - vendor/
```

Breaking down .gitlab-ci.yml

- This lesson is a rapid and very high-level overview of Python; the following lessons provide more detail.
- You may find it valuable to return to this lesson and work through the appropriate examples as a review after you read about the features covered in subsequent lessons.
- If this lesson was mostly a review for you, or if you'd like to learn more about only a few features, feel free to jump around, using the index or table of contents.

artifacts – paths

- Much like with caching, you specify an array of paths, which can include wildcards like so:

```
package:  
  artifacts:  
    paths:  
      - public/  
      - tests/*.html
```

artifacts – name

- By default, any uploaded files will be stored in a .zip archive titled artifacts.zip.
- You can use the name parameter to change the default filename (it will still end with .zip), as demonstrated in the following code:

```
package:  
artifacts:  
  name: cool_project_name
```

The variables parameter

- You can store variables per job or globally using the variables keyword.
- The value for variables should be an array of key/value pairs that can be represented by strings or integers (for both key and value), although typically the key will be an all-capitalized string for ease of recognition.
- Any variables used should be of a non-sensitive nature; they are not considered an appropriate method for storing secrets.
- Locally defined variables will override their globally defined counterparts, but to have a job run with no access to globally defined variables, you should redeclare it with an empty array like so:

```
cool_job_name:  
  variables: {}
```

Adding .gitlab-ci.yml to our example project

before_script:

- apt-get update -qq && DEBIAN_FRONTEND=noninteractive apt-get install -y -qq ca-certificates git php php-xml
- php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
- php composer-setup.php
- php composer.phar install

phpunit:

script:

- vendor/bin/phpunit tests/ROT13FormatterTest

Adding .gitlab-ci.yml to our example project

Run untagged jobs

Indicates whether this runner can pick jobs without tags

Now, when you go to Project | CI/CD | Pipelines, you should see your new pipeline there, ready to go:



Adding .gitlab-ci.yml to our example project

passed Pipeline #6 triggered 16 minutes ago by  Administrator

adding git install to CI/CD startup so Composer works

⌚ 1 job from [2-add-testing-and-ci](#) in 2 minutes 9 seconds (queued for 2 seconds)

-o a8319230 ... 📄

Pipeline Jobs 1

Test

 [phpunit](#) 

- You can click the status (passed, in the preceding screenshot) to receive a breakdown of the pipeline. Yours should look similar to this:

Adding .gitlab-ci.yml to our example project

Adam O'Grady > monoROT13 > Jobs > #3



Job #3 triggered 1 minute ago by Adam O'Grady

```
Running with gitlab-runner 11.1.0 (081978aa)
on gitlab-runner-0 21e117ae
Using Docker executor with image ubuntu:latest ...
Pulling docker image ubuntu:latest ...
Using docker image sha256:cd6d8154f1e16e38493c3c2798977c5e142be5e5d41403ca89883840c6d51762 for ubuntu:latest ...
Running on runner-21e117ae-project-2-concurrent-0 via gitlab-runner-0...
Cloning repository...
Cloning into '/builds/judges119/monoROT13'...
Checking out a62246f2 as 2-add-testing-and-ci...
Skipping Git submodules setup
$ apt-get update -qq && apt-get install -y -qq php
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package perl-modules-5.26.
```

Deconstructing an advanced .gitlab-ci.yml file

- The header of the .gitlab-ci.yml file defines an image to use from the Docker Registry (image: bitnami/laravel:latest), as well as defining a service (in this case, the postgres:9.6 image, also from the Docker registry).
- This service container will be attached directly to our Laravel image to provide a database without needing to set up a database as part of a before_script each time a job is run:
`image: bitnami/laravel:latest`

`services:`

`- postgres:9.6`

Deconstructing an advanced .gitlab-ci.yml file

- The variables that are declared all relate to our database and will be used in the jobs so that we can connect to our database service image.
- Of particular note is the DB_HOST parameter, which simply specifies postgres as the hostname.
- This is because with GitLab CI, any services that are connected are referred to by their image name, in this case, postgres:
`variables:`
 `POSTGRES_DATABASE: postgres`
 `POSTGRES_PASSWORD: password`
 `DB_HOST: postgres`
 `DB_USERNAME: root`

Deconstructing an advanced .gitlab-ci.yml file

- Next, we define three stages that we want: test, package, and deploy.
- These stages will be run sequentially (and any errors in an earlier stage will cause the whole pipeline to be canceled):

`stages:`

- `test`
- `package`
- `deploy`

Deconstructing an advanced .gitlab-ci.yml file

- The big difference between the two is that `php_unit_test` also contains a cache section, which means that after the job is finished, the entire `vendor/` directory will be zipped up and stored under the composer key for reuse in future jobs and pipelines.
- This helps reduce the time taken to build and execute subsequent runs of this job and any other jobs that use the same cache.
- Once both jobs are finished (and both are successful), the next stage can begin:

```
php_unit_test:  
  stage: test  
  script:  
    - cp .env.example .env  
    - composer install  
    - php artisan key:generate  
    - php artisan migrate  
    - vendor/bin/phpunit  
  cache:  
    key: composer  
    paths:  
      - vendor/  
  
js_unit_test:  
  stage: test  
  script:  
    - npm install  
    - npm run test
```

Deconstructing an advanced .gitlab-ci.yml file

- For this example, we've also set the policy to pull to demonstrate it.
- This means that while it will download any available cache stored under the composer key, it will not re-upload any files once the job has finished.
- This task will also only operate on tagged commits, which is handy if you're doing versioned releases of software.
- Once all of the scripts have been executed, all of the files in the public/ directory will be packaged and uploaded as artifacts and stored for up to a week:

```
package_upload:  
  stage: package  
  script:  
    - composer install  
    - npm install  
    - webpack  
  cache:  
    key: composer  
    paths:  
      - vendor/  
  policy: pull  
artifacts:  
  paths:  
    - public/  
  expire_in: 1 week  
only:  
  - tags
```

Deconstructing an advanced .gitlab-ci.yml file

```
deploy_production:
  stage: deploy
  script:
    - composer install
    - npm install
    - webpack
    - .composer/vendor/bin/envoy run deploy
  environment:
    name: production
    url: http://192.168.0.1
  when: manual
  only:
    - master
```

GitLab CI/CD web UI

The screenshot shows the GitLab CI/CD web interface for the project `monoROT13`. The left sidebar has a navigation menu with items like Project, Repository, Issues, Merge Requests, CI / CD (which is currently selected), Pipelines, Jobs, Schedules, Charts, Operations, Wiki, Snippets, and Settings. The main area displays a table of pipelines. The table columns are Status, Pipeline, Commit, and Stages. The table shows the following data:

Status	Pipeline	Commit	Stages
skipped	#9 by latest	3-trial-env... -o c43d82bc testing the manual keyword	00:02:07 3 days ago
passed	#8 by latest	3-trial-env... -o 5908db68 #3 trialling environments i...	00:02:07 3 days ago
passed	#7 by latest	master -o 0b7cb642 Merge branch '2-add-testi...	00:02:06 3 days ago
passed	#6 by latest	2-add-testi... -o a8319230 adding git install to CI/CD ...	00:02:09 1 week ago
failed	#5 by latest	2-add-testi... -o 0406088e adding php-xml package t...	00:00:55 1 week ago
failed	#4 by latest	2-add-testi... -o d7a00f3f adding ca-certificates pac...	00:00:52 1 week ago

At the top right, there are buttons for "Run Pipeline", "Clear Runner Caches", and "CI Lint". The top navigation bar includes links for GitLab, Projects, Groups, Activity, Milestones, Snippets, and a search bar.

GitLab CI/CD web UI

Check your .gitlab-ci.yml

- The output of a linted .gitlab-ci.yml file looks like this:

Content of .gitlab-ci.yml

```
1 image: bitnami/laravel:latest
2
3 services:
4   - postgres:9.6
5
6 variables:
7   POSTGRES_DATABASE: postgres
8   POSTGRES_PASSWORD: password
9   DB_HOST: postgres
10  DB_USERNAME: root
11
12 stages:
13   - test
14   - package
15   - deploy
16
17 php_unit_test:
18   stage: test
19   script:
20     - cp .env.example .env
21     - composer install
22     - php artisan key:generate
23     - php artisan migrate
24     - vendor/bin/phpunit
25 cache:
```

Validate

Clear

Status: syntax is correct

GitLab CI/CD web UI

- If the syntax passes its validation, you'll be shown the preceding message, as well as a table (visible in the following screenshot) that breaks down the keys and values provided in the script. You can use these to sanity check your work:

Parameter	Value
Test Job - php_unit_test	<p>Tag list: Only policy: Except policy: Environment: When: on_success</p> <pre>cp .env.example .env composer install php artisan key:generate php artisan migrate vendor/bin/phpunit</pre>
Test Job - js_unit_test	<p>Tag list: Only policy: Except policy: Environment: When: on_success</p> <pre>npm install npm run test</pre>

GitLab CI/CD web UI

The screenshot shows the GitLab CI/CD web interface for the project `monoROT13`. The left sidebar has a navigation menu with options: Project, Repository, Issues (1), Merge Requests (0), CI / CD (selected), Pipelines, Jobs (selected), Schedules, Charts, Operations, Wiki, and Snippets. The main content area displays a table of CI jobs. The table has columns: Status, Job, Pipeline, Stage, Name, and Coverage. There are 11 total jobs, with 0 pending, 0 running, and 10 finished. The finished jobs are listed below:

Status	Job	Pipeline	Stage	Name	Coverage
manual	#11 3-trial-env... -o c43d82bc allowed to fail manual	#9 by 🐱	test	phpunit	
passed	#10 3-trial-env... -o 5908db68	#8 by 🐱	test	phpunit	⌚ 02:07 🕒 3 days ago
passed	#9 3-trial-env... -o 5908db68	#8 by 🐱	test	phpunit	⌚ 02:07 🕒 3 days ago
passed	#8 master -o 0b7cb642	#7 by 🐱	test	phpunit	⌚ 02:06 🕒 3 days ago
passed	#7 2-add-testi... -o a8319230	#6 by 🐱	test	phpunit	⌚ 02:09 🕒 1 week ago
failed	#6 2-add-testi... -o 0406088e	#5 by 🐱	test	phpunit	⌚ 00:55 🕒 1 week ago
failed	#5 2-add-testi... -o d7a00f3f	#4 by 🐱	test	phpunit	⌚ 00:52 🕒 1 week ago

GitLab CI/CD web UI

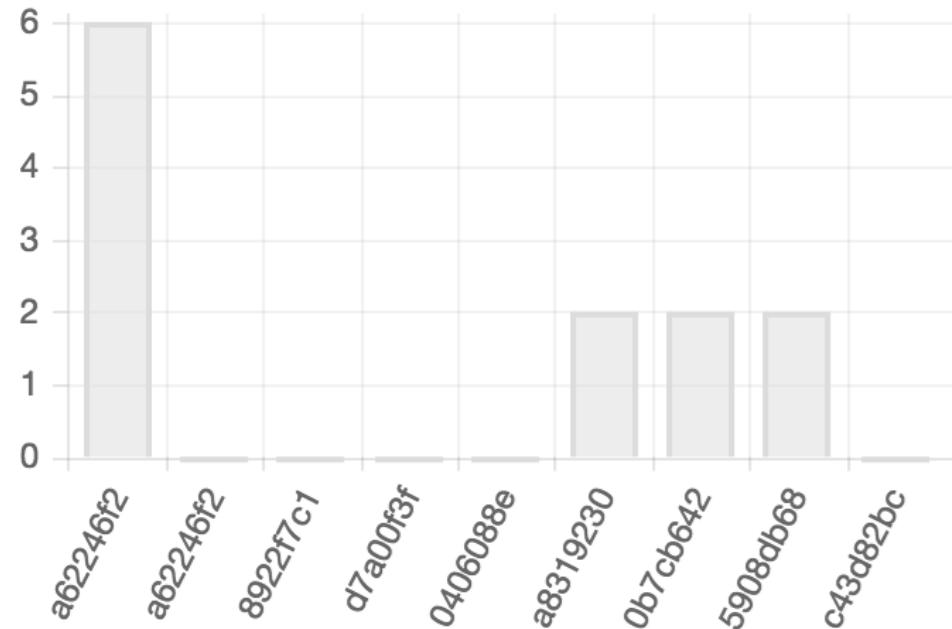
- From the menu on the left, under CI/CD, you can click Charts to get some graphical information about the CI/CD process attached to this project:

A collection of graphs regarding Continuous Integration

Overall statistics

- Total: **9 pipelines**
- Successful: **3 pipelines**
- Failed: **4 pipelines**
- Success ratio: **42%**

Commit duration in minutes for last 30 commits



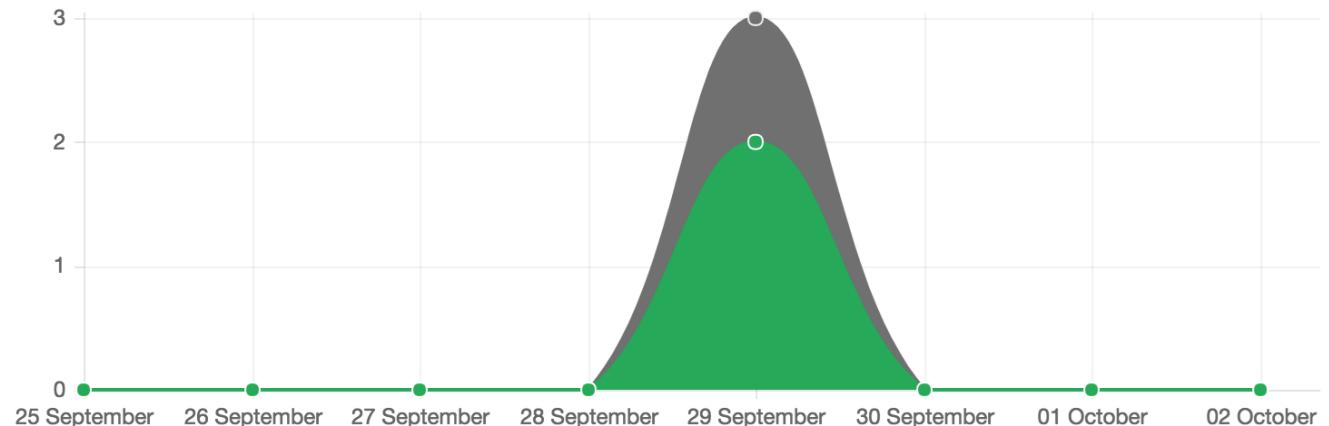
GitLab CI/CD web UI

- Next up, we have charts showing the breakdown of pipelines over the past week, month, and year:

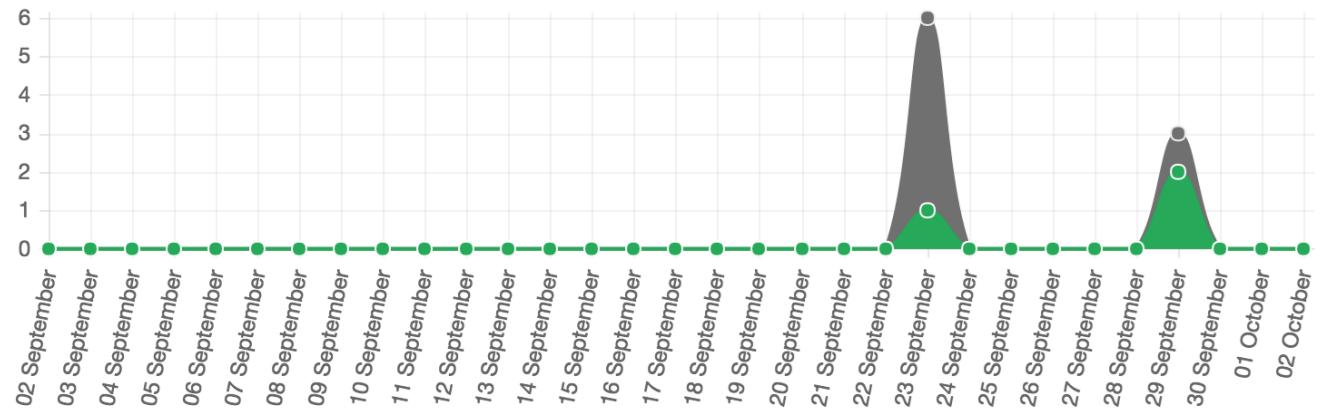
Pipelines charts

● success ● all

Pipelines for last week (25 Sep - 2 Oct)



Pipelines for last month (2 Sep - 2 Oct)



GitLab CI/CD web UI

GitLab Projects Groups Activity Milestones Snippets 🔧 + This project Search 🌐 0 2 🔍 2 📋 🎯 💀

M monoROT13 Adam O'Grady > monoROT13 > Pipelines > Environments

Project Available 1 Stopped 0 New environment

Repository Issues Merge Requests CI / CD Operations Environments Kubernetes

Environment	Deployment	Job	Commit	Updated
testing	#3 by 🚗	phpunit #12	→ e38afdea 💀 removing the manual keyw...	4 minutes ago

Re-deploy

Summary

- By now, you should have a firm grip on the basics of continuous integration and continuous delivery, as done by GitLab.
- We started out in this lesson by exploring the concept of CI/CD and giving you a crash course in it.
- Next up, we ran through the process of installing a GitLab Runner, Don't forget that the Runner is the platform where your CI/CD stages are actually executed; you can have many of them per project or per GitLab installation to help parallelize the work.
- We looked at their installation on Ubuntu and CentOS, and manually installing one via a binary.



"Complete Lab 12 & 13"

The scope of runners ALL TIERS

Runners are available based on who you want to have access:

- Shared runners are available to all groups and projects in a GitLab instance.
- Group runners are available to all projects and subgroups in a group.
- Project runners are associated with specific projects. Typically, project runners are used by one project at a time.

Shared runners

- Shared runners are available to every project in a GitLab instance.
- Use shared runners when you have multiple jobs with similar requirements. Rather than having multiple runners idling for many projects, you can have a few runners that handle multiple projects.

If you are using a self-managed instance of GitLab:

- Your administrator can install and register shared runners by going to your project's Settings > CI/CD, expanding Runners, and selecting Show runner installation instructions. These instructions are also available in the documentation.

Enable shared runners for a project

For existing projects, an administrator must install and register them.

To enable shared runners for a project:

- On the top bar, select Main menu > Projects and find your project.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- Turn on the Enable shared runners for this project toggle.

Enable shared runners for a group

To enable shared runners for a group:

- On the top bar, select Main menu > Groups and find your group.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- Turn on the Enable shared runners for this group toggle.

Disable shared runners for a project

To disable shared runners for a project:

- On the top bar, select Main menu > Projects and find your project.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- In the Shared runners area, turn off the Enable shared runners for this project toggle.

Shared runners are automatically disabled for a project:

- If the shared runners setting for the parent group is disabled, and
- If overriding this setting is not permitted at the project level.

Disable shared runners for a group

To disable shared runners for a group:

- On the top bar, select Main menu > Groups and find your group.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- Turn off the Enable shared runners for this group toggle.
- Optional. To allow shared runners to be enabled for individual projects or subgroups, select Allow projects and subgroups to override the group setting.

How shared runners pick jobs

- Shared runners process jobs by using a fair usage queue. This queue prevents projects from creating hundreds of jobs and using all available shared runner resources.
- The fair usage queue algorithm assigns jobs based on the projects that have the fewest number of jobs already running on shared runners.
- For example, if these jobs are in the queue:

Job 1 for Project 1

Job 2 for Project 1

Job 3 for Project 1

Job 4 for Project 2

Job 5 for Project 2

Job 6 for Project 3

Group runners

- Use group runners when you want all projects in a group to have access to a set of runners.
- Group runners process jobs by using a first in, first out (FIFO) queue.

Create a group runner

- You can create a group runner for your self-managed GitLab instance or for GitLab.com. You must have the Owner role for the group.

To create a group runner:

- Install GitLab Runner.
- On the top bar, select Main menu > Groups and find your group.
- On the left sidebar, select CI/CD > Runners.
- In the upper-right corner, select Register a group runner.
- Select Show runner installation and registration instructions. These instructions include the token, URL, and a command to register a runner.

Project runners

Use project runners when you want to use runners for specific projects.

For example, when you have:

- Jobs with specific requirements, like a deploy job that requires credentials.
- Projects with a lot of CI activity that can benefit from being separate from other runners.

You can set up a project runner to be used by multiple projects. Project runners must be enabled for each project explicitly.

- Project runners process jobs by using a first in, first out (FIFO) queue.

Create a project runner

To create a project runner:

- Install GitLab Runner.
- On the top bar, select Main menu > Projects and find the project where you want to use the runner.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- In the Project runners section, note the URL and token.
- Register the runner.
- The runner is now enabled for the project

Enable a project runner for a different project

After a project runner is created, you can enable it for other projects.

- Prerequisites: You must have at least the Maintainer role for:
 - The project where the runner is already enabled.
 - The project where you want to enable the runner.
 - The project runner must not be locked

Prevent a project runner from being enabled for other projects

To lock or unlock a project runner:

- On the top bar, select Main menu > Projects and find the project where you want to enable the runner.
- On the left sidebar, select Settings > CI/CD.
- Expand Runners.
- Find the project runner you want to lock or unlock. Make sure it's enabled. You cannot lock shared or group runners.
- Select Edit () .
- Select the Lock to current projects checkbox.
- Select Save changes.

Executors ALL TIERS

To jump into the specific documentation for each executor, visit:

- SSH
- Shell
- Parallels
- VirtualBox
- Docker
- Docker Machine (auto-scaling)
- Kubernetes
- Custom

Prerequisites for non-Docker executors

- Executors that do not rely on a helper image require a Git installation on the target machine and in the PATH.
- Always use the latest available version of Git.
- GitLab Runner uses the git lfs command if Git LFS is installed on the target machine.
- Ensure Git LFS is up to date on any systems where GitLab Runner uses these executors.

Selecting the executor

Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Clean build environment for every build	✗	✗	✓	✓	✓	✓	conditional (4)
Reuse previous clone if it exists	✓	✓	✗	✗	✓	✗	conditional (4)
Runner file system access protected (5)	✓	✗	✓	✓	✓	✓	conditional
Migrate runner machine	✗	✗	partial	partial	✓	✓	✓
Zero-configuration support for concurrent builds	✗	✗ (1)	✓	✓	✓	✓	conditional (4)
Complicated build environments	✗	✗ (2)	✓ (3)	✓ (3)	✓	✓	✓
Debugging build problems	easy	easy	hard	hard	medium	medium	medium

Compatibility chart

- Supported features by different executors:

Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Secure Variables	✓	✓	✓	✓	✓	✓	✓
GitLab Runner Exec command	✗	✓	✗	✗	✓	✓	✓
.gitlab-ci.yml: image	✗	✗	✓ (1)	✓ (1)	✓	✓	✓ (via \$CUSTOM_ENV_CI_JOB_IMA GE)
.gitlab-ci.yml: services	✗	✗	✗	✗	✓	✓	✓
.gitlab-ci.yml: cache	✓	✓	✓	✓	✓	✓	✓
.gitlab-ci.yml: artifacts	✓	✓	✓	✓	✓	✓	✓
Passing artifacts between stages	✓	✓	✓	✓	✓	✓	✓
Use GitLab Container Registry private images	n/a	n/a	n/a	n/a	✓	✓	n/a
Interactive Web terminal	✗	✓ (UNIX)	✗	✗	✓	✓	✗

Compatibility chart

- Supported systems by different shells:

Shells	Bash	PowerShell Desktop	PowerShell Core	Windows Batch (deprecated)
Windows	✗ (4)	✓ (3)	✓	✓ (2)
Linux	✓ (1)	✗	✓	✗
macOS	✓ (1)	✗	✓	✗
FreeBSD	✓ (1)	✗	✗	✗

Compatibility chart

- Supported systems for interactive web terminals by different shells:

Shells	Bash	PowerShell Desktop	PowerShell Core	Windows Batch (deprecated)
Windows	✗	✗	✗	✗
Linux	✓	✗	✗	✗
macOS	✓	✗	✗	✗
FreeBSD	✓	✗	✗	✗

Plan and operate a fleet of shared runners

When you host a fleet of shared runners, you need a well-planned infrastructure that takes into consideration your:

- Computing capacity.
- Storage capacity.
- Network bandwidth and throughput.
- Type of jobs (including programming language, OS platform, and dependent libraries).
- Use this guide to develop a GitLab Runner deployment strategy based on your organization's requirements.

Consider your workload and environment

Before you deploy runners, consider your workload and environment requirements.

- Create a list of the teams that you plan to onboard to GitLab.
- Catalog the programming languages, web frameworks, and libraries in use at your organization. For example, GoLang, C++, PHP, Java, Python, JavaScript, React, Node.js.
- Estimate the number of CI/CD jobs each team may execute per hour, per day.
- Validate if any team has build environment requirements that cannot be addressed by using containers.

Workers, executors, and autoscaling capabilities

- The gitlab-runner executable runs your CI/CD jobs.
- Each runner is an isolated process that picks up requests for job executions and deals with them according to pre-defined configurations.
- As an isolated process, each runner can create “sub-processes” (also called “workers”) to run jobs.

Basic configuration: one runner, one worker

- After the installation is complete, you execute the runner registration command just once and you select the shell executor.
- Then you edit the runner config.toml file to set concurrency to 1.

```
concurrent = 1
```

```
[[runners]]  
  name = "instance-level-runner-001"  
  url = ""  
  token = ""  
  executor = "shell"
```

Intermediate configuration: one runner, multiple workers

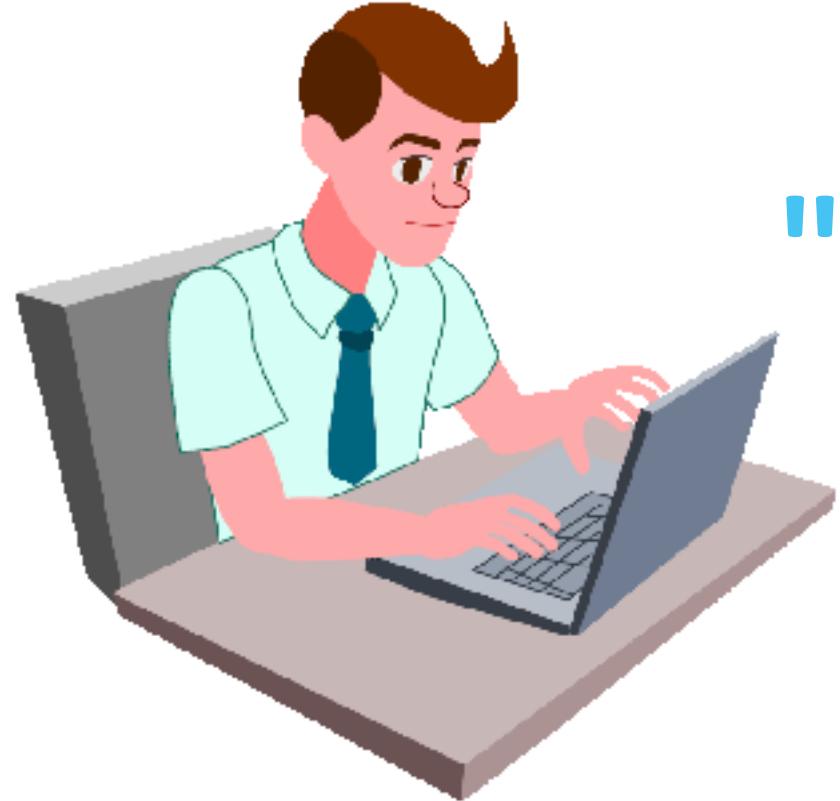
- You can also register multiple runner workers on the same machine.
- When you do this, the runner's config.toml file has multiple [[runners]] sections in it.
- If all of the additional runner workers are registered to use the shell executor, and you update the value of the global configuration option, concurrent, to 3, the upper limit of jobs that can run concurrently on this host is equal to three.

```
concurrent = 3

[[runners]]
  name = "instance_level_shell_001"
  url = ""
  token = ""
  executor = "shell"

[[runners]]
  name = "instance_level_shell_002"
  url = ""
  token = ""
  executor = "shell"

[[runners]]
  name = "instance_level_shell_003"
  url = ""
  token = ""
  executor = "shell"
```



"Complete Lab 14 & 15"