

# Git – Version Control

Roi Yehoshua  
2018

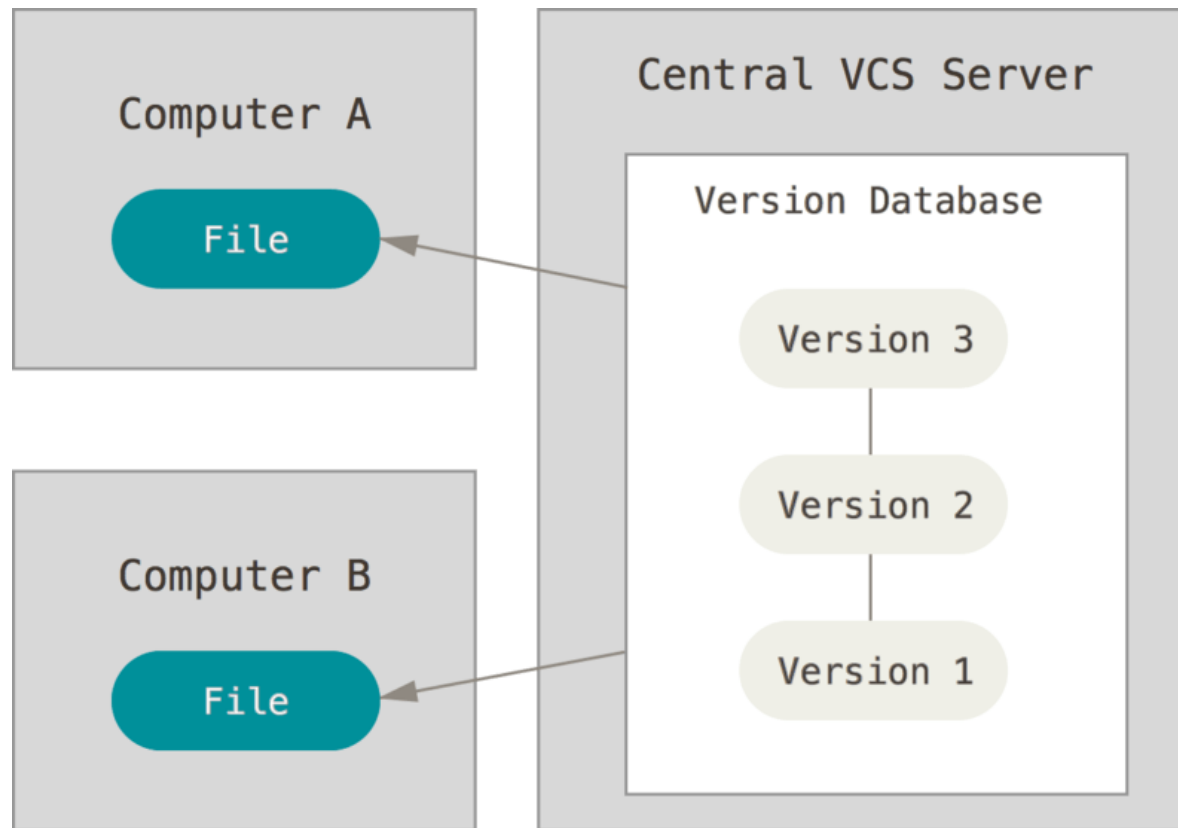
# Version Control Systems (VCS)

---

- ▶ Version control is a system that records changes to a set of files over time so that you can recall specific versions later
- ▶ There are two types of version control systems:
  - ▶ In **centralized version control systems**, developers use a shared single repository
    - ▶ Example: Subversion (SVN), CVS
  - ▶ In **distributed version control systems**, each developer works directly with his own local repository, and changes are shared between repositories as a separate step
    - ▶ Example: GIT

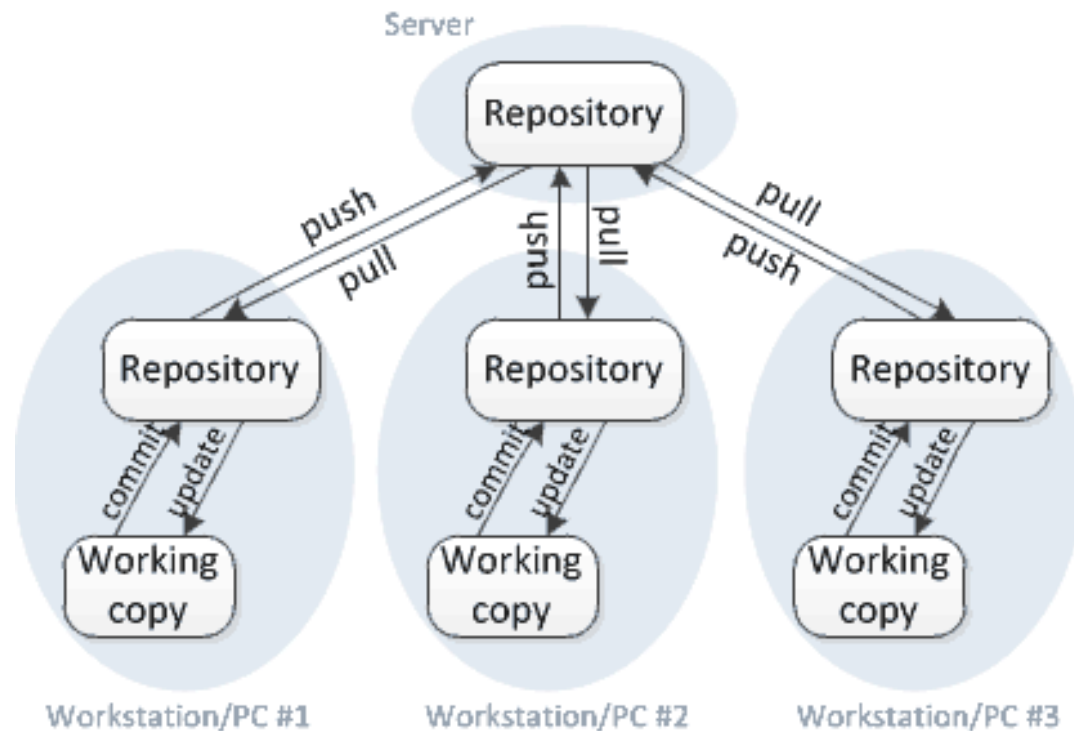
# Centralized Version Control Systems

- ▶ A single server contains all the versioned files
- ▶ A number of clients can check out files from that central place



# Distributed Version Control

- ▶ Every computer contains a full-fledged repository, independent of network access or a central server
- ▶ If any server dies, any of the client repositories can be copied back up to the server





# Git

---

- ▶ Git is the most popular version control system
  - ▶ 42.9% of professional software developers use Git as their primary source control system<sup>[</sup>
- ▶ A distributed version control system
  - ▶ Every computer has its own local repository
  - ▶ Has all the information
  - ▶ Every developer has a copy of the same repository
- ▶ Git was created by Linus Torvalds in 2005 for development of the Linux kernel
- ▶ There are many Git servers that can host your system
  - ▶ e.g, GitHub, Bitbucket, GitLab
- ▶ The command line is the only place you can run all Git commands
  - ▶ Most of the GUIs implement only a partial subset of Git functionality for simplicity

# Git Servers

---



Free Plans	Public Repos	Private Repos	Collaborators	Storage Space	Hosting	Support
GitHub Public	Unlimited	0	Unlimited	N / A	Cloud	Email / Forum
Bitbucket Small teams	Unlimited	Unlimited (1Gb /project)	5	N / A	Cloud	Email / Forum
GitLab Cloud Hosted	Unlimited	Unlimited (10Gb / Project)	Unlimited	Unlimited	Cloud	Forum
GitLab Community Edition	Unlimited	Unlimited	Unlimited	N / A	Self-hosted	Forum
Coding Free Plan	Unlimited	Unlimited	10	1GB	Cloud	Email /Forum

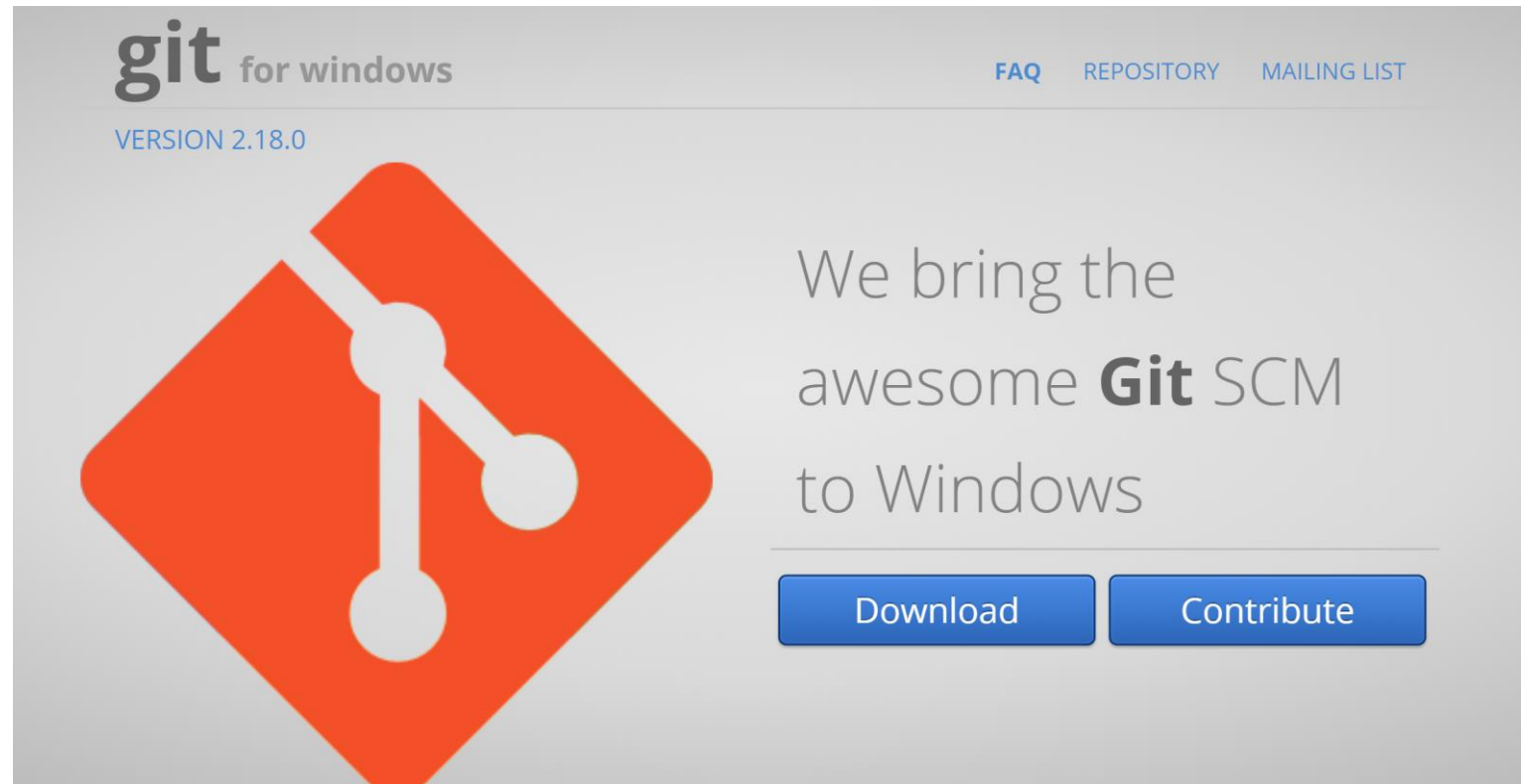
# Installing Git

---

- ▶ Before you start using Git, you have to make it available on your computer
- ▶ Even if it's already installed, it's probably a good idea to update to the latest version
  - ▶ Latest version is 2.15.0
- ▶ Installing on Windows:
  - ▶ Download the latest [Git for Windows installer](#)
  - ▶ When you've successfully started the installer, you should see the **Git Setup** wizard screen. Follow the **Next** and **Finish** prompts to complete the installation.
  - ▶ The default options are pretty sensible for most users.

# Installing Git For Windows

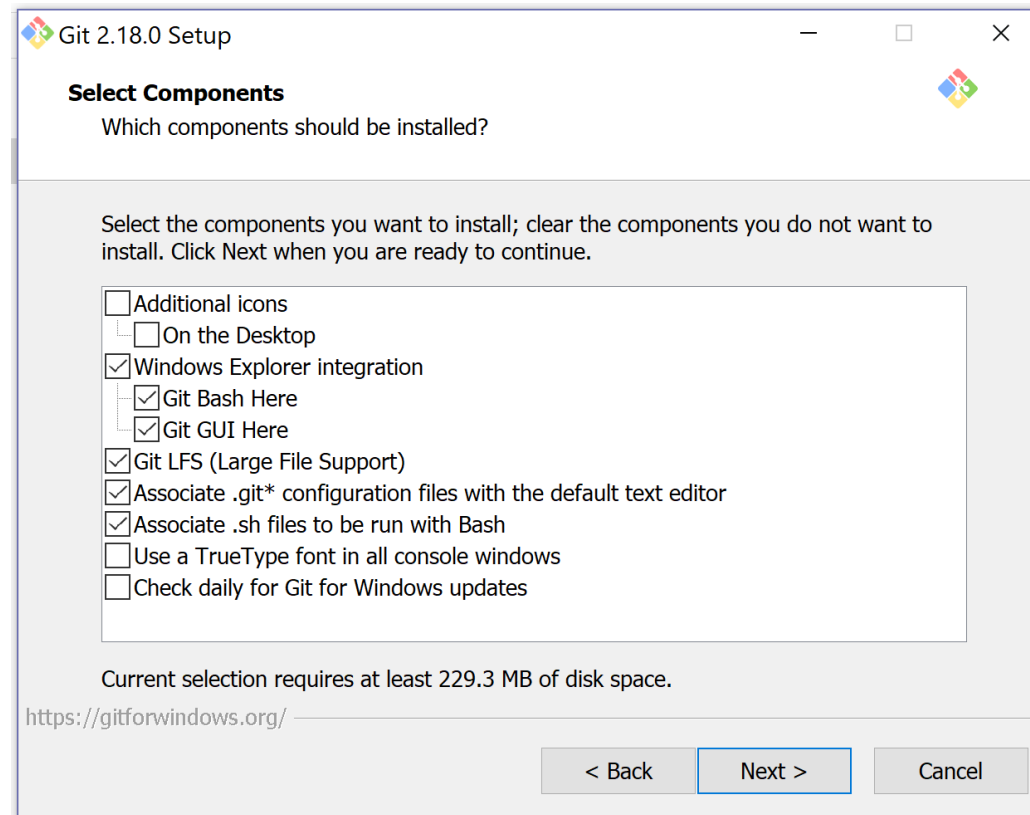
---





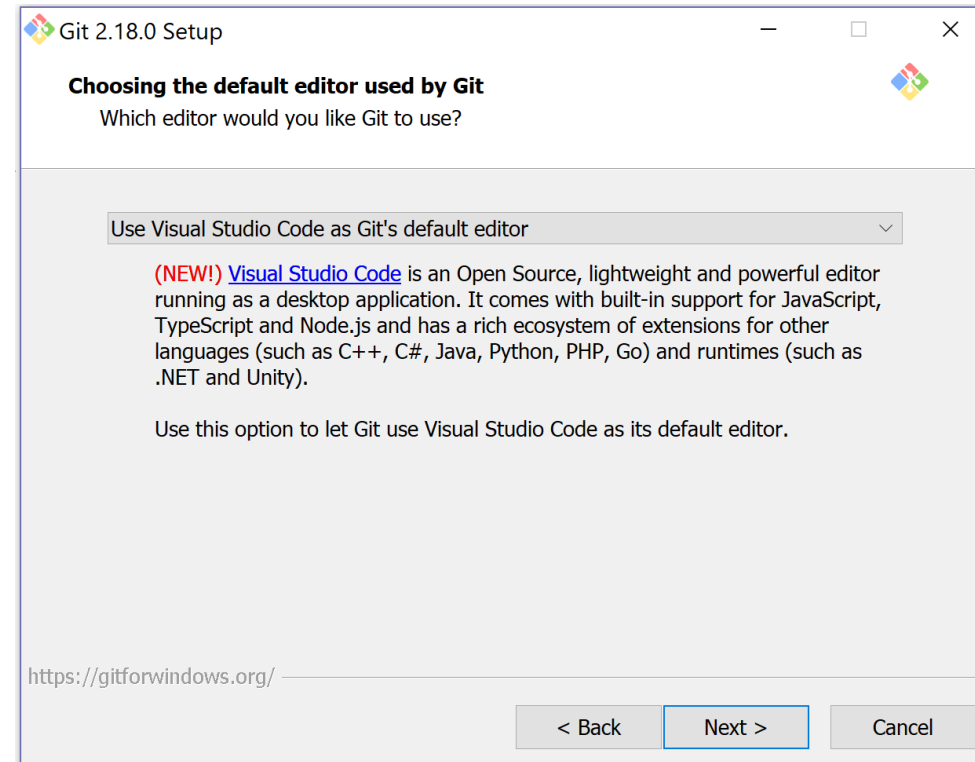
# Installing Git For Windows

---



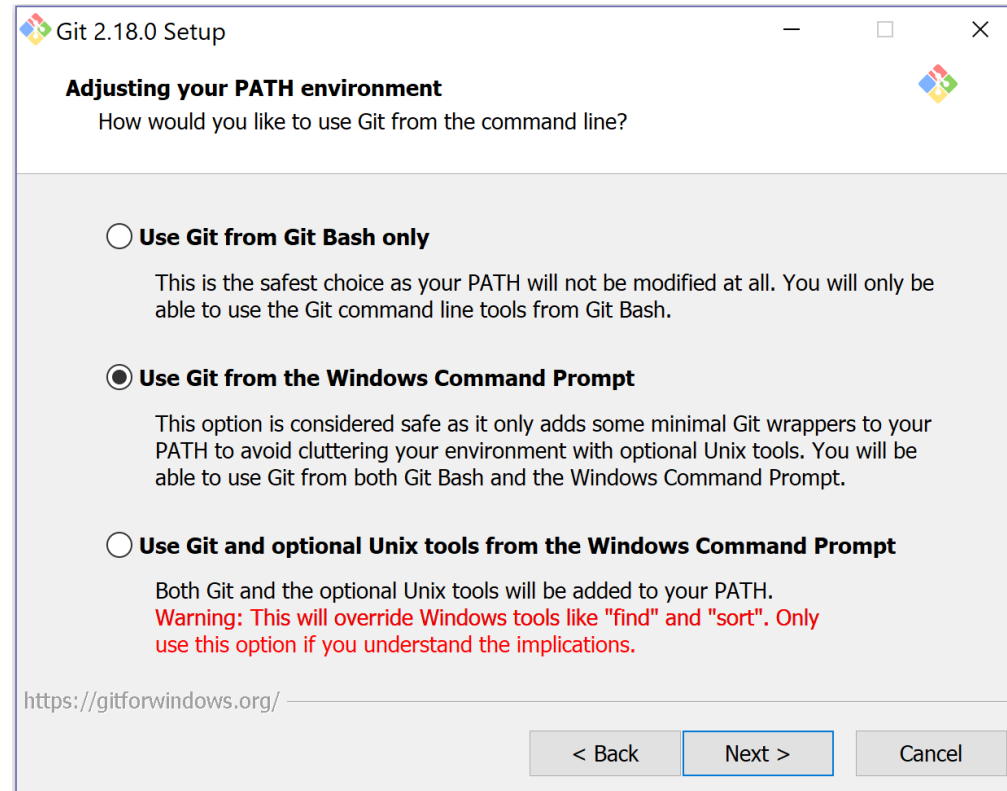
# Installing Git For Windows

---



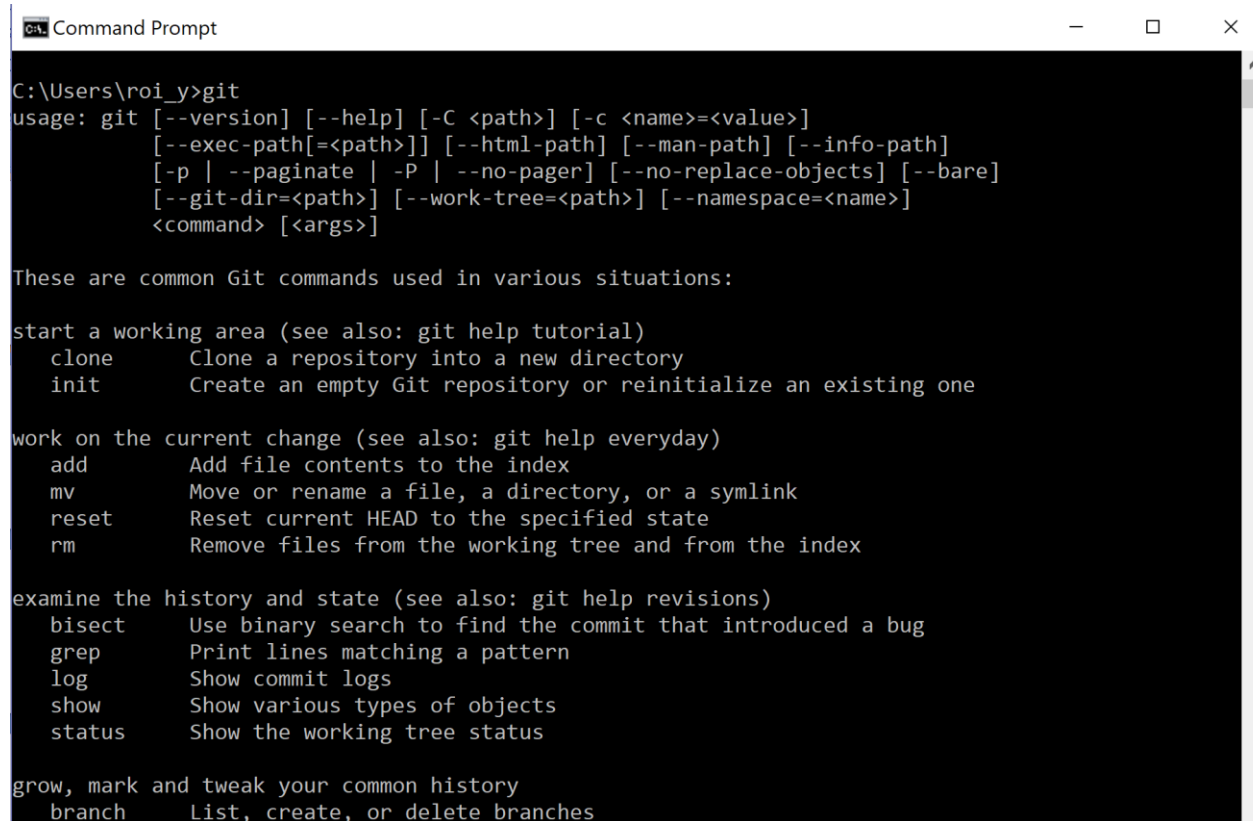
# Installing Git For Windows

---



# Git Bash

- ▶ Open a command prompt
- ▶ Type git to check the installation



```
Command Prompt
C:\Users\roi_y>git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | -P | --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
```

# First-Time Git Setup

---

- ▶ The **git config** command lets you get and set configuration variables that control all aspects of how Git looks and operates
- ▶ The first thing you should do when you install Git is to set your user name and email address
- ▶ This is important because every Git commit uses this information

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

- ▶ You need to do this only once if you pass the --global option
- ▶ If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find

# Getting Help

---

- ▶ If you ever need help while using Git, you can use the `git help` command for the full manpage help or ask for the more concise help with the `-h` option, as in:

```
C:\Users\roi_y>git add -h
usage: git add [<options>] [--] <pathspec>...

-n, --dry-run          dry run
-v, --verbose          be verbose

-i, --interactive      interactive picking
-p, --patch            select hunks interactively
-e, --edit             edit current diff and apply
-f, --force            allow adding otherwise ignored files
-u, --update           update tracked files
--renormalize          renormalize EOL of tracked files (implies -u)
-N, --intent-to-add    record only the fact that the path will be added later
-A, --all              add changes from all tracked and untracked files
--ignore-removal       ignore paths removed in the working tree (same as --no-all)
--refresh              don't add, only refresh the index
--ignore-errors        just skip files which cannot be added because of errors
--ignore-missing        check if - even missing - files are ignored in dry run
--chmod <(+/-)x>      override the executable bit of the listed files
```

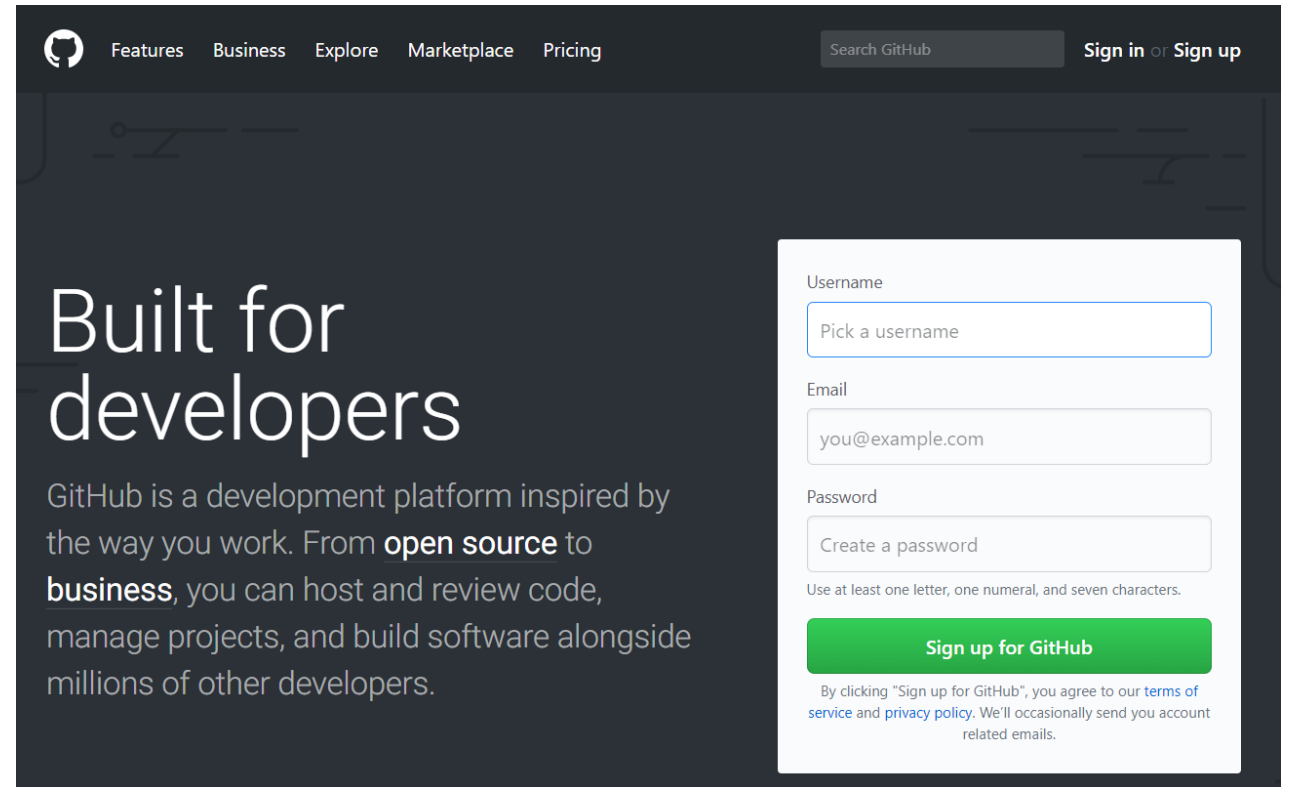
# Getting a Git Repository

---

- ▶ You typically obtain a Git repository in one of two ways:
  - ▶ You can take a local directory that is currently not under version control, and turn it into a Git repository
  - ▶ You can **clone** an existing Git repository from elsewhere
- ▶ In either case, you end up with a Git repository on your local machine, ready for work

# Creating a Repository on GitHub

- ▶ Go to <https://github.com/>
- ▶ Sign up as a new user
- ▶ Registration is free



The screenshot shows the GitHub homepage with a dark theme. The navigation bar at the top includes links for Features, Business, Explore, Marketplace, and Pricing, along with a search bar and links to Sign in or Sign up. The main content area features the text "Built for developers" and a description of GitHub as a development platform. On the right side, there is a white sign-up form with fields for Username, Email, and Password, and a green "Sign up for GitHub" button.

Username  
Pick a username

Email  
you@example.com

Password  
Create a password

Use at least one letter, one numeral, and seven characters.

[Sign up for GitHub](#)

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We'll occasionally send you account related emails.



# Creating a Repository on GitHub


## Create a new repository

A repository contains all the files for your project, including the revision history.


---

Owner

Repository name

 roiyeho

 / 


test 

Great repository names are short and memorable. Need inspiration? How about [scaling-palm-tree](#).


Description (optional)

A test repository

---

☒  **Public**

Anyone can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.


---

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None**

 | 

Add a license: **None** 

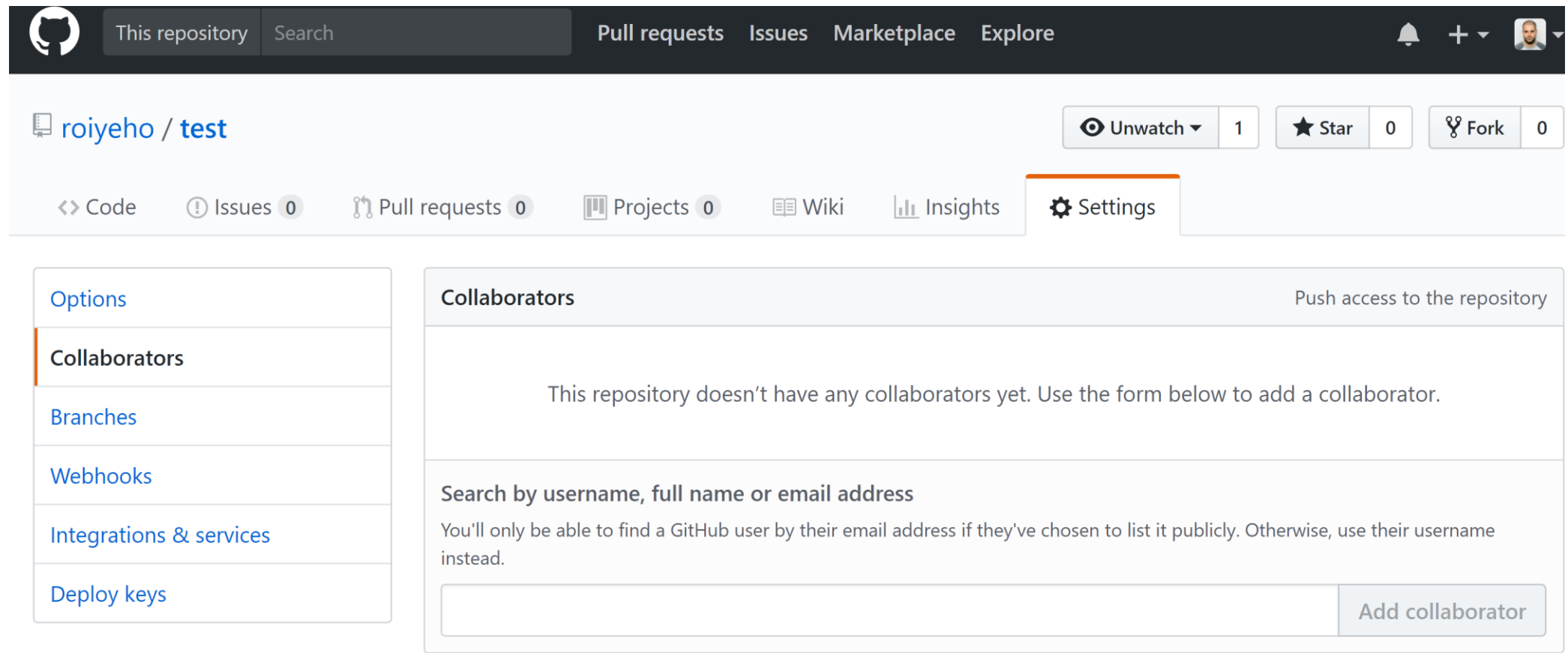
---

Create repository

- ▶ The url of your repository will be `https://github.com/[user name]/[repository name]`
  - ▶ e.g., `https://github.com/roiyeho/test`

# Adding Collaborators

- ▶ Under Settings -> Collaborators you can add collaborators to your repository



The screenshot shows the GitHub interface for a repository named 'test' by user 'roiyeho'. The top navigation bar includes links for 'This repository', 'Search', 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right, there are icons for notifications, a dropdown menu, and a user profile. Below the navigation bar, the repository name 'roiyeho / test' is displayed, followed by buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). A secondary navigation bar contains links for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Insights', and 'Settings' (which is highlighted with an orange bar). On the left side of the 'Settings' page, a sidebar lists various settings: 'Options', 'Collaborators' (highlighted with an orange bar), 'Branches', 'Webhooks', 'Integrations & services', and 'Deploy keys'. The main content area of the 'Collaborators' tab is titled 'Collaborators' and includes a link 'Push access to the repository'. It contains a message: 'This repository doesn't have any collaborators yet. Use the form below to add a collaborator.' Below this message, there is a search instruction: 'Search by username, full name or email address'. A note states: 'You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.' At the bottom, there is a text input field and an 'Add collaborator' button.

# Initialize Repository From Existing Code

---

- ▶ If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory

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

- ▶ and type:

```
$ git init
```

- ▶ This creates a new subdirectory named **.git** that contains all of your necessary repository files — a Git repository skeleton
- ▶ To stop tracking our project with git, just remove this directory

```
$ rm -rf .git
```

# Cloning an Existing Repository

---

- ▶ If you want to get a copy of an existing Git repository use **git clone <url>**
- ▶ git clone receives a full copy of nearly all data that the server has
- ▶ git clone implicitly adds the **origin** remote for you
- ▶ For example, to clone the repository we created previously, type:

```
$ git clone https://github.com/roiyeho/test
```

```
C:\git>git clone https://github.com/roiyeho/test
Cloning into 'test'...
warning: You appear to have cloned an empty repository.

C:\git>cd test

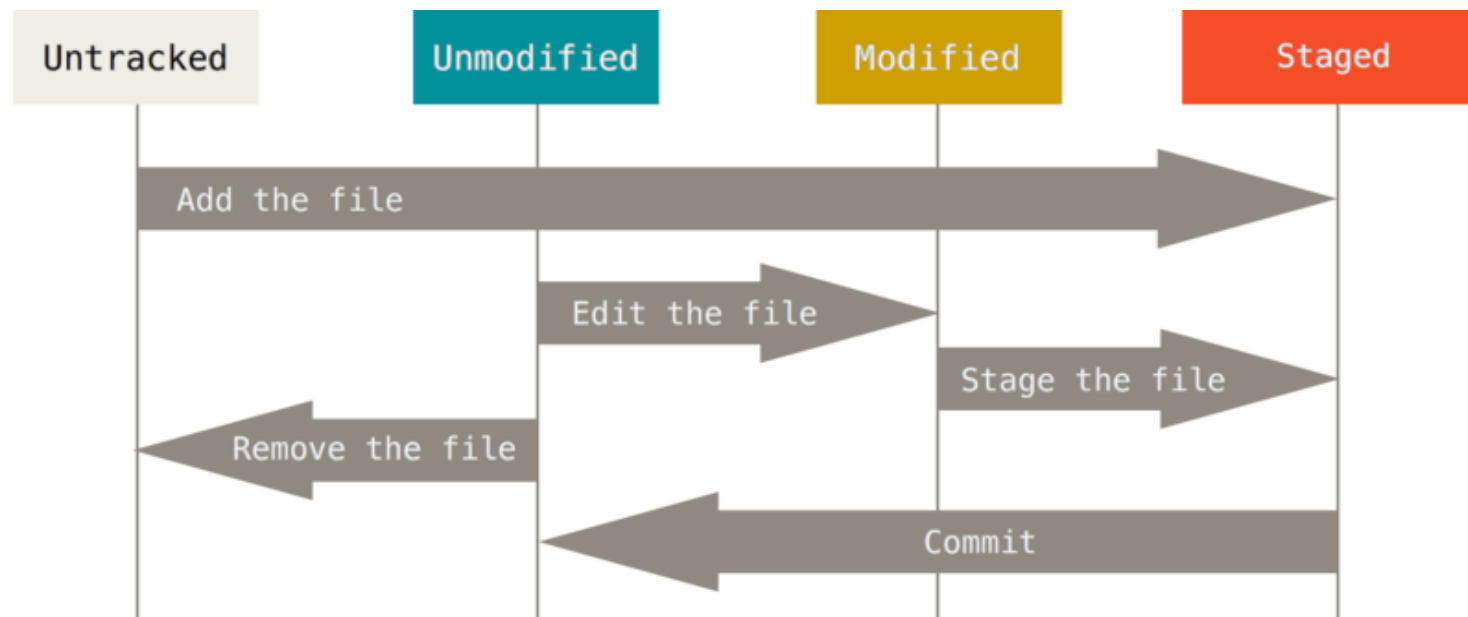
C:\git\test>dir /a
Volume in drive C is OS
Volume Serial Number is 825C-2D89

Directory of C:\git\test

16/07/2018  11:46    <DIR>        .
16/07/2018  11:46    <DIR>        ..
16/07/2018  11:46    <DIR>        .git
               0 File(s)              0 bytes
               3 Dir(s)  5,024,522,240 bytes free
```

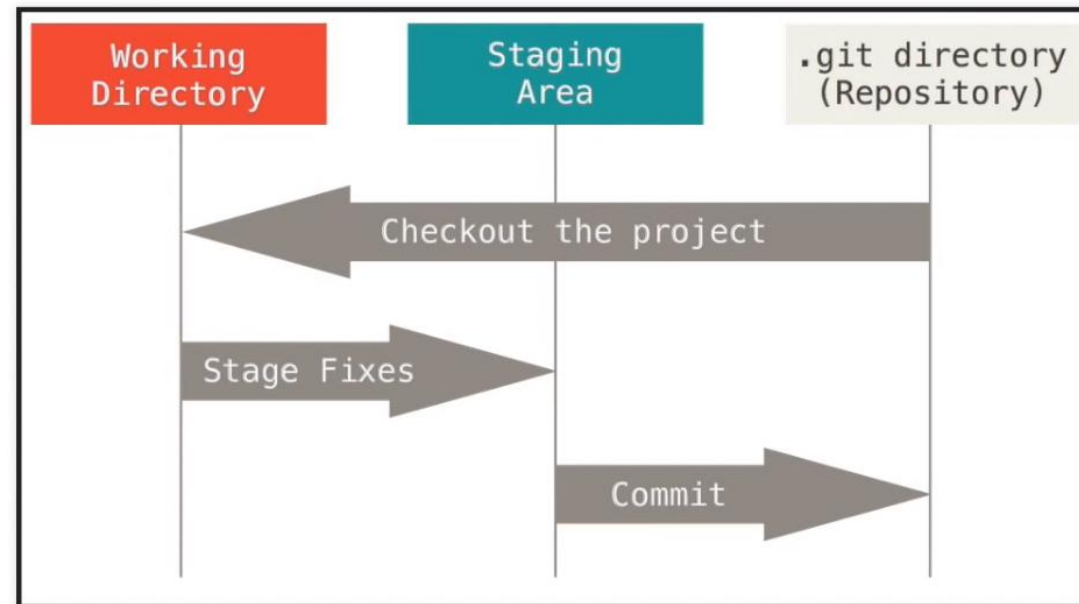
# The Three States

- ▶ Git has three main states that your files can reside in:
  - ▶ **Modified** – you've changed the file but have not committed it to your database yet
  - ▶ **Staged** – you've marked a modified file in its current version to go into your next commit
  - ▶ **Committed** – the data is safely stored in your local database



# Git Sections

- ▶ There are three main sections of a Git project
  - ▶ **The Git directory** – where Git stores the metadata and object database for your project
  - ▶ **The working tree** – a single checkout of one version of the project
  - ▶ **The staging area** - a file, generally contained in your Git directory, that stores information about what will go into your next commit



# Checking the Status of the Files

---

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

```
C:\git\test>git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

- ▶ This means you have a clean working directory
- ▶ Git also doesn't see any untracked files, or they would be listed here
- ▶ Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server.
  - ▶ For now, that branch is always "master", which is the default

# Checking the Status of the Files

---

- ▶ Let's say you add a new file to your project, a simple README file.
- ▶ Now when you run **git status**, you see your untracked file like so:

```
C:\git\test>echo 'My Project' > README

C:\git\test>git status
On branch master

No commits yet

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

        README

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

- ▶ You can see that your new README file is untracked
- ▶ Untracked means that Git sees a file you didn't have in the previous snapshot (commit)
- ▶ It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include



# Tracking New Files

---

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

```
$ git add README
```

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

```
C:\git\test>git add README  
  
C:\git\test>git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
  
    new file:   README
```

- ▶ git add command takes a path name for either a file or a directory
  - ▶ If it's a directory, the command adds all the files in that directory recursively

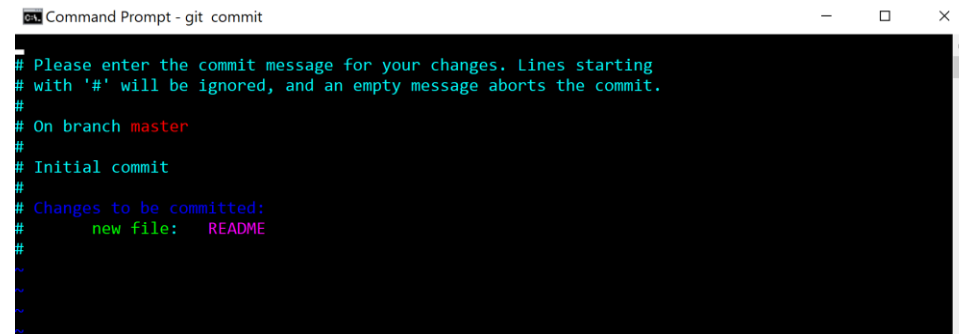
# Committing Your Changes

---

- ▶ The simplest way to commit is to type **git commit**:

```
$ git commit
```

- ▶ Doing so launches your editor of choice
  - ▶ defined by the shell's EDITOR environment variable
- ▶ The editor displays the following text:



```
Command Prompt - git commit
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   README
#
~
~
~
```

- ▶ When you exit the editor Git creates your commit with that commit message (with the comments and diff stripped out)

# Committing Your Changes

---

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

```
$ git commit -m "Added a README file"
```

```
C:\git\test>git commit -m "Added a README file"
[master (root-commit) af5c5c1] Added a README file
1 file changed, 1 insertion(+)
create mode 100644 README
```

- ▶ Now you've created your first commit!
- ▶ You can see that the commit has given you some output about itself: which branch you committed to (master), how many files were changed, and statistics about lines added and removed in the commit

# Staging a Modified File

---

- ▶ Let's change the README file
- ▶ If you change a previously tracked file and then run your git status command again, you get something that looks like this:

```
C:\git\test>git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

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

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

- ▶ The README has been modified in the working directory but not yet staged
- ▶ To stage it, you run the **git add** command

# Staging a Modified File

---

- ▶ To stage the file, run the **git add** command

```
C:\git\test>git add README

C:\git\test>git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README
```

# Viewing Your Staged and Unstaged Changes

---

- ▶ If you want to know exactly what you changed, not just which files were changed — you can use the **git diff** command
- ▶ To see what you've changed but not yet staged, type **git diff** with no arguments
- ▶ If you want to see what you've staged that will go into your next commit, you can use **git diff --staged**

```
C:\git\test>git diff --staged
diff --git a/README b/README
index af257a1..edee6ba 100644
--- a/README
+++ b/README
@@ -1,1 @@
-'My Project'
+'My Project' V2
```

# Viewing the Commit History

---

- ▶ The **git log** command shows the existing commit history

```
C:\git\test>git log
commit af5c5c1c9b5c583dc9efa01280d6c8e278c4f18f (HEAD -> master)
Author: Roi Yehoshua <roiyehe@gmail.com>
Date:   Mon Jul 16 11:54:11 2018 +0300

    Added a README file
```

- ▶ The format option allows you to specify your own log output format, e.g.:

```
viki@c3po: ~/test
viki@c3po:~/test$ git log --pretty=format:"%h - %an, %ar : %s"
4839fc0 - Roi Yehoshua, 4 minutes ago : Added project description to README
ba90d73 - Roi Yehoshua, 20 minutes ago : Added a README file
viki@c3po:~/test$
```

# Pushing Changes to Remote Repository

---

- ▶ When you have your project at a point that you want to share, you have to push it upstream
- ▶ The command for this is **git push <remote> <branch>**
- ▶ If you want to push your master branch to your origin server, then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

```
C:\git\test>git push origin master
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 462 bytes | 231.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/roiyeho/test
 * [new branch]      master -> master
```



# Pushing Changes to Remote Repository

---

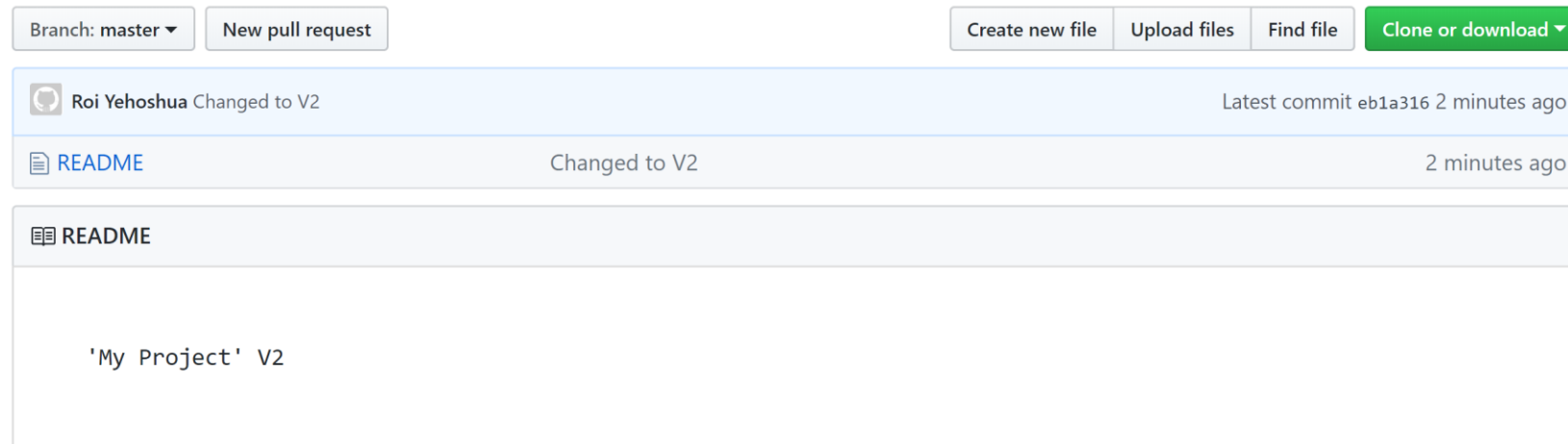
- ▶ Now when you run **git status**, you can see that your local repository is up-to-date with the remote repository and you have a clean directory:

```
C:\git\test>git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

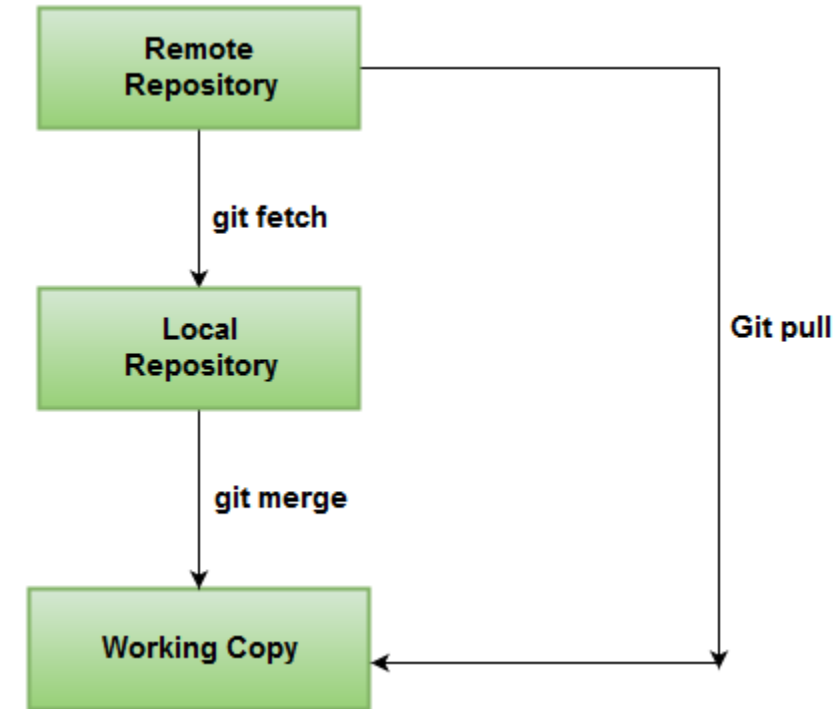
# Inspecting the Remote Repository

- ▶ Go to your GitHub repository url to inspect the changes:



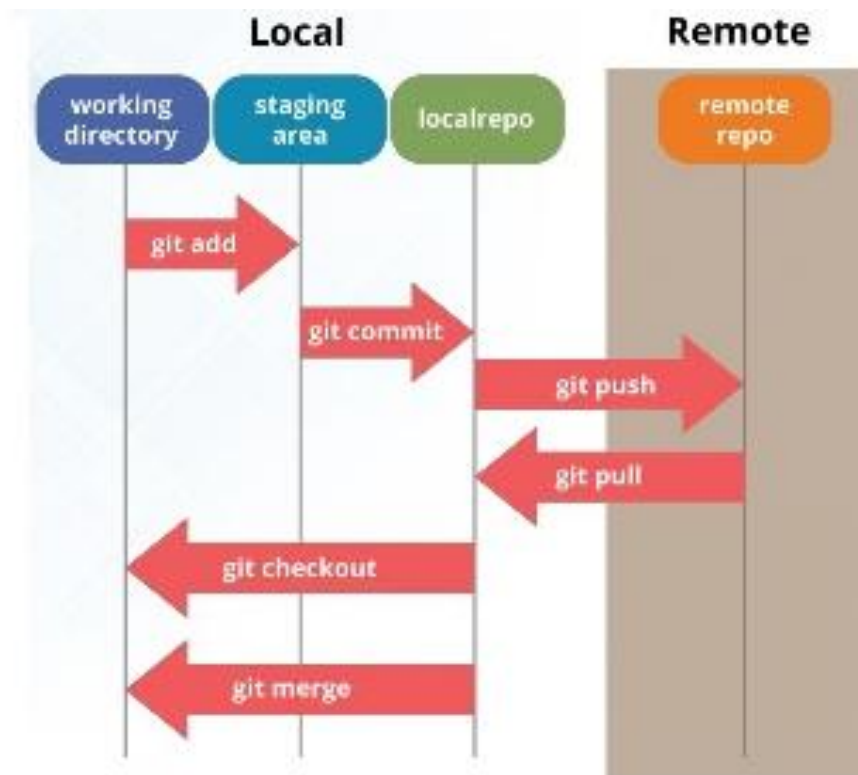
# Fetching and Pulling Changes

- ▶ Other developers might have been pushing changes to the remote repository while you've been working on your local repository
- ▶ **git fetch** - gathers any commits from the server and stores them in your local repository (without merging)
- ▶ To integrate the commits into your working directory use **git merge**
- ▶ **git pull** – does a git fetch followed by a git merge



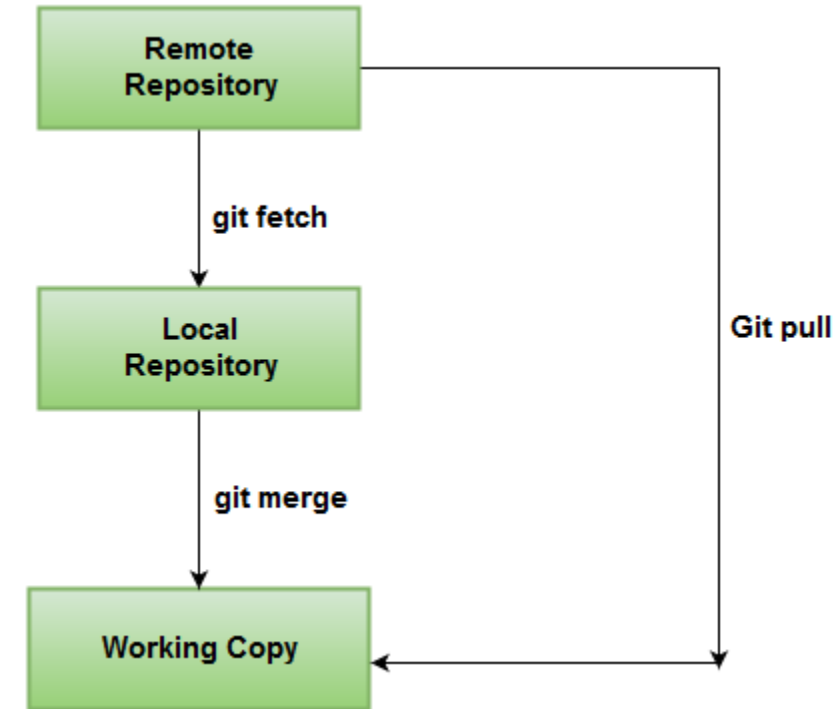
# Git Commands Summary

---



# Fetching and Pulling Changes

- ▶ Other developers might have been pushing changes to the remote repository while you've been working on your local repository
- ▶ **git fetch** - gathers any commits from the server and stores them in your local repository (without merging)
- ▶ To integrate the commits into your working directory use **git merge**
- ▶ **git pull** – does a git fetch followed by a git merge



# Git Branches

---

- ▶ Branching means you can diverge from the main line of development and continue to do work without messing with that main line
- ▶ A repository typically contains several branches
  - ▶ A branch is a version of your repository.
- ▶ Out of the scope for this course
- ▶ You can find more details at
  - ▶ <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

# Resolving a Merge Conflict

---

- ▶ 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
- ▶ For example, if you and another person both edited the file *styleguide.md* on the same lines in different branches of the same Git repository, you'll get a merge conflict error when you try to merge these branches
- ▶ You must resolve this merge conflict with a new commit before you can merge these branches

# Resolving a Merge Conflict

---

- ▶ Navigate into the local Git repository that has the merge conflict
- ▶ Generate a list of the files affected by the merge conflict
- ▶ In this example, the file *styleguide.md* has a merge conflict

```
$ 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")
```

- ▶ Open your favorite text editor and navigate to the file that has merge conflicts



# Resolving a Merge Conflict

---

- ▶ To see the beginning of the merge conflict in your file, search the file for the conflict marker <<<<<<
- ▶ When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line <<<<<< HEAD
- ▶ Next, you'll see =====, which divides your changes from the changes in the other branch, followed by >>>>>> BRANCH-NAME
- ▶ 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
```

# Resolving a Merge Conflict

---

- ▶ Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches
- ▶ Delete the conflict markers <<<<<<, =====, >>>>>> and make the changes you want in the final merge
- ▶ Add or stage your changes by using **git add**

```
$ git add .
```

- ▶ Commit your changes with a comment.

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