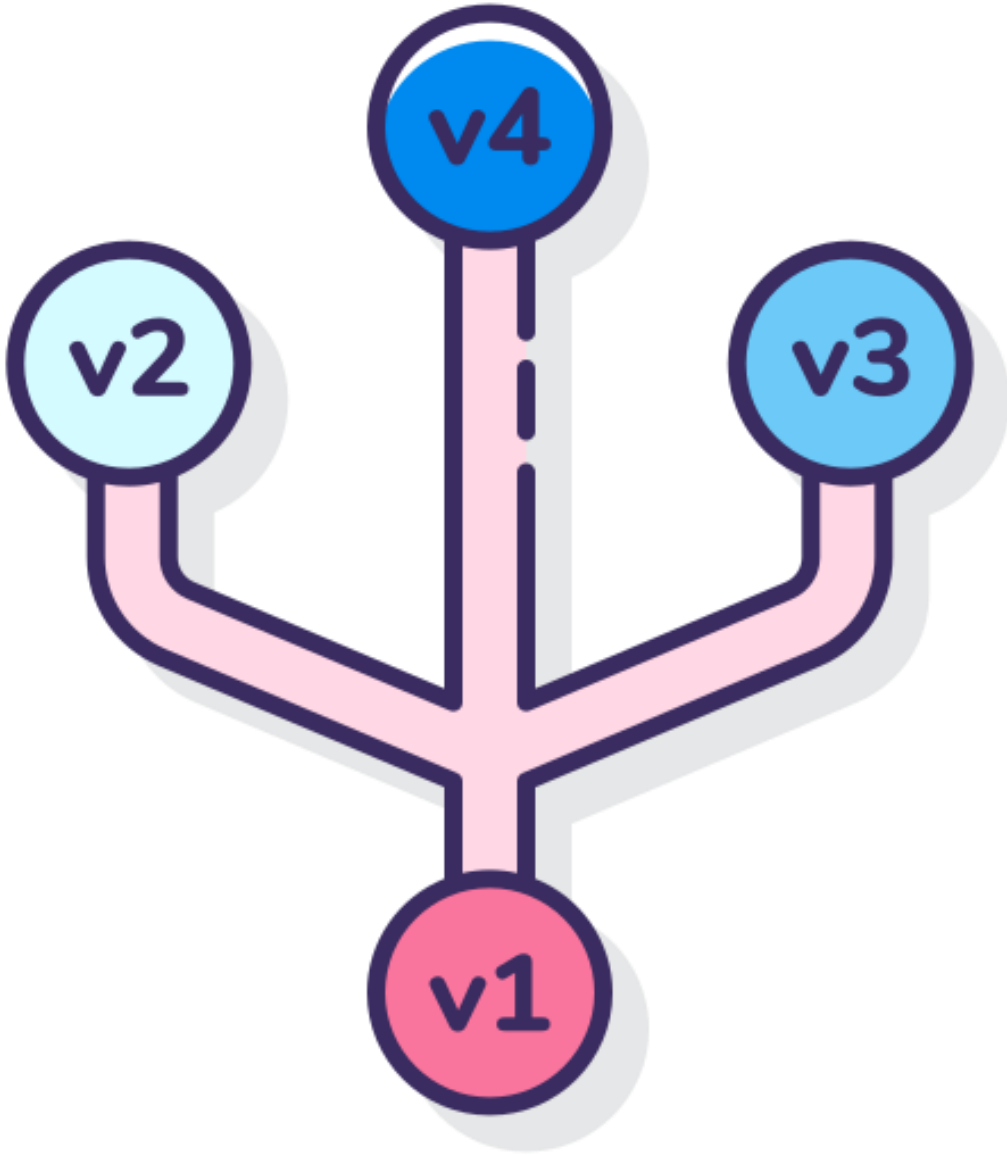
Introduction to **Git**

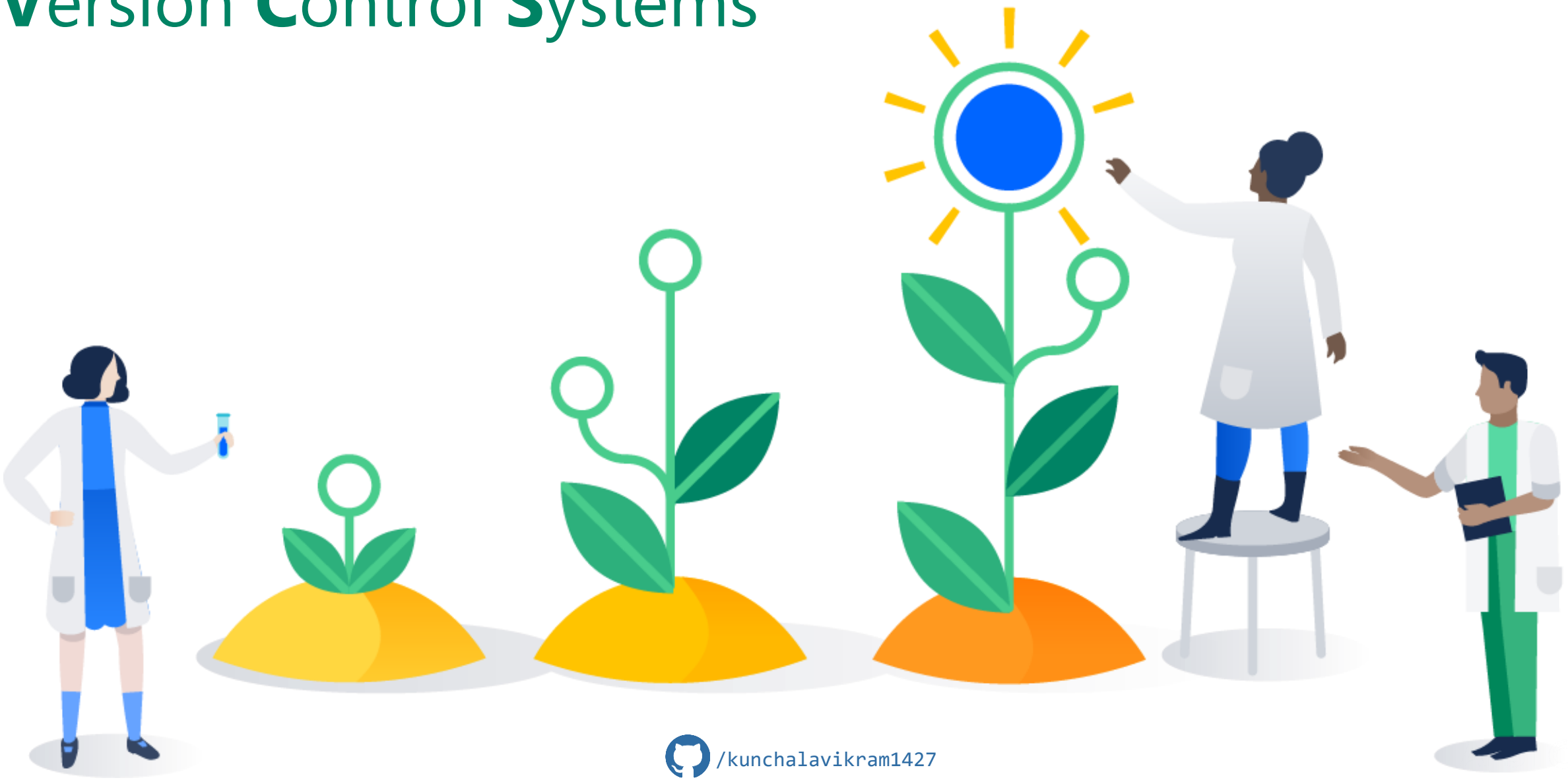
Distributed Version Control



Vikram
IoT Application Dev & DevOps

Contents

Version **C**ontrol **S**ystems



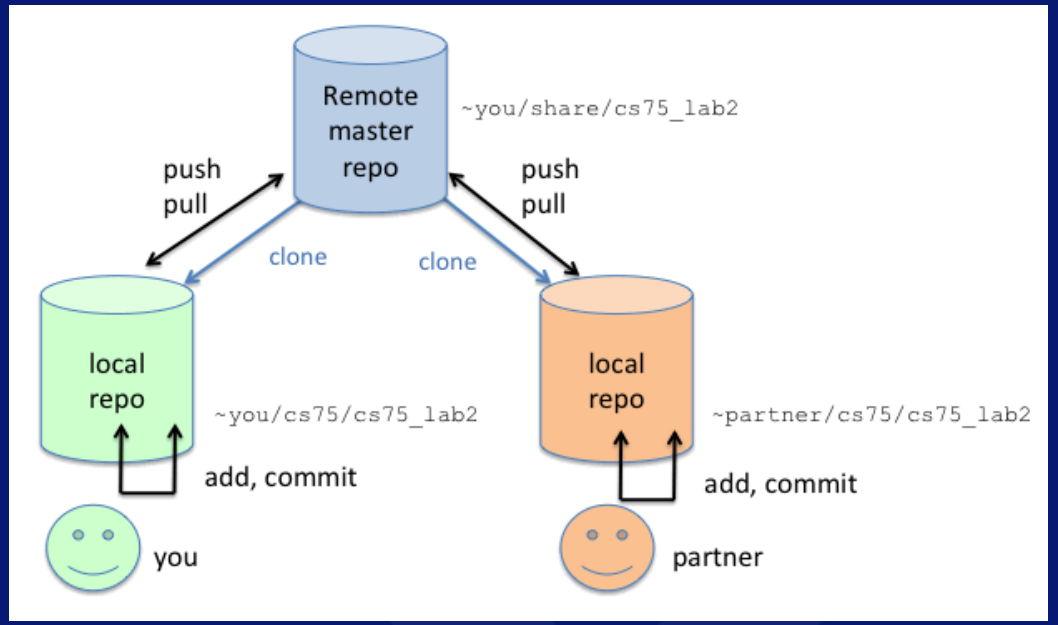
Version Control Systems(VCS)

- Version control, also known as source control, is the practice of tracking and managing changes to software code.
- It enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team.
- These systems are critical to ensure everyone has access to the latest code. As development gets more complex, there's a bigger need to manage multiple versions of entire products.
- Version control also enables one person you to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. In rare cases, when two people make conflicting edits to the same line of a file, then the version control system requests human assistance in deciding what to do.
- Version control gives access to historical versions of your project. If you make a mistake, you can roll back to a previous version. You can reproduce and understand a bug report on a past version of your software. You can also undo specific edits without losing all the work that was done in the meanwhile. For any part of a file, you can determine when, why, and by whom it was ever edited.
- In DevOps, other than keeping track of changes, VCS also helps in developing and shipping the products faster.

Version Control Systems git:kunchalavikram1427

Repositories and working copies

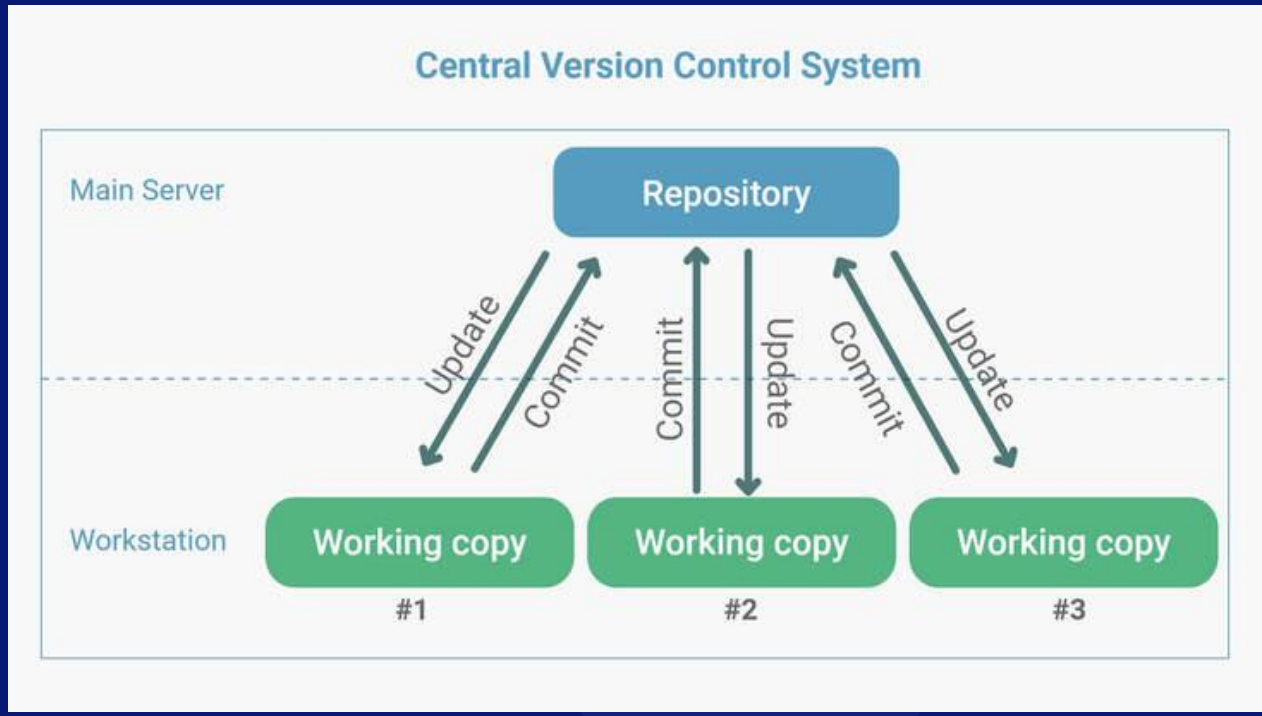
- Version control uses a remote repository and a working copy where you do your work
- Your working copy is your personal copy of all the files in the project. You make arbitrary edits to this copy, without affecting your teammates. When you are happy with your edits, you commit your changes to a repository
- A repository is a database of all the edits to, and/or historical versions (snapshots) of, your project
- It is possible for the repository to contain edits(from other developers) that have not yet been applied to your working copy
- You can update your working copy to incorporate any new edits or versions that have been added to the repository since the last time you updated(pulling the latest changes)



Central Vs Distributed VCS

Central VCS

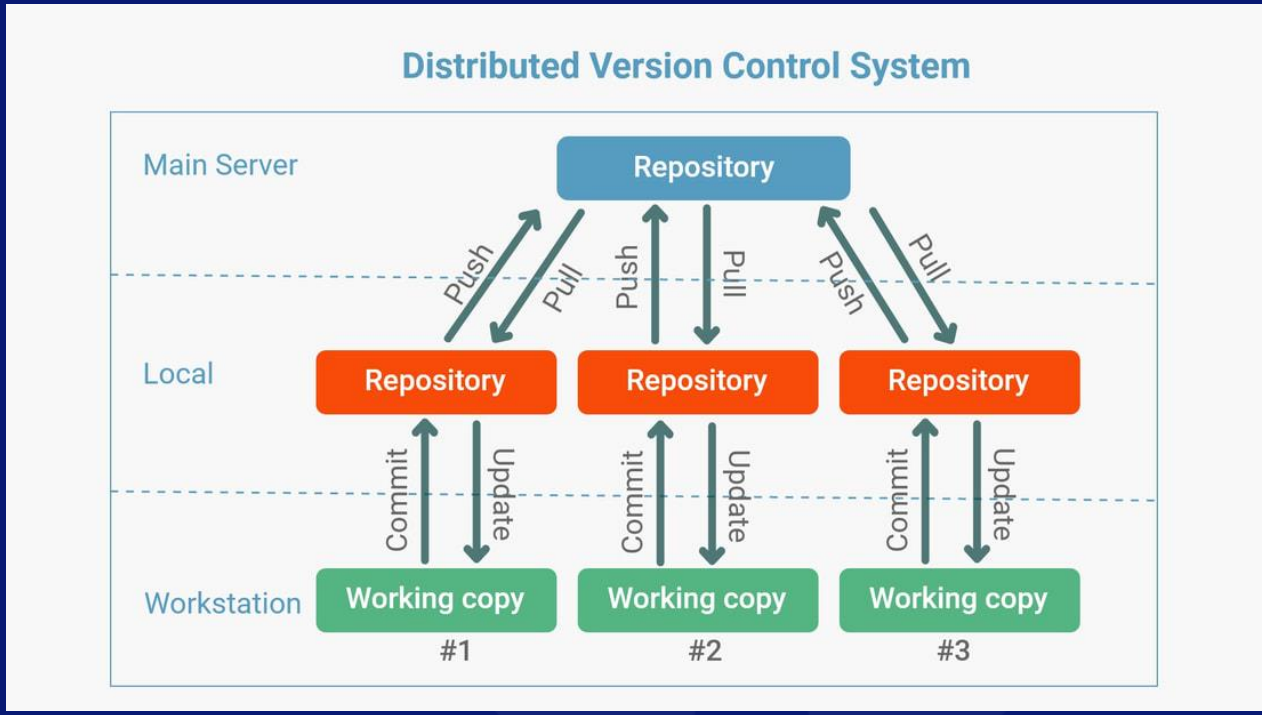
- The main difference between centralized and distributed version control is the number of repositories
- In centralized version control, there is just one repository, and in distributed version control, there are multiple repositories
- In CVCS, the central server stores all the data. Each user gets his or her own working copy, but there is just one central repository. As soon as you commit, it is possible for your co-workers to update and to see your changes
- If the central server gets crashed, there is a high chance of losing the data
- For every command, CVCS connects the central server which impacts speed of operation
- Ex: Subversion VCS



Central Vs Distributed VCS

Distributed VCS

- In distributed version control, each user gets his or her own local repository and a remote repository for all
- After you commit to the local repository, others have no access to your changes until you push your changes to the central repository
- If other users want to check your changes, they will pull the updated central repository to their local repository, and then they update in their local copy
- Even if the main server crashes, code that is in the local systems can be used to restore the data
- DVCS is fast compared to CVCS because you don't have to contact the central server for every command
- Ex: Git, Mercurial



git

Installing Git

Git

Installing Git

For Windows

- Visit https://git-scm.com/book/en/v2/Getting-Started-Installing-Git to install Git for your operating systems
- To install GUI clients, visit https://git-scm.com/downloads/guis
- Verify if Git is installed by running the command `git version` in the CMD
- Use `git help` to get the list of supported commands and `git help <sub-command>` for help on that command

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git version
git version 2.27.0.windows.1
```

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git help
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
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a bug
   diff              Show changes between commits, commit and working tree, etc
   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
   commit            Record changes to the repository
   merge             Join two or more development histories together
   rebase            Reapply commits on top of another base tip
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   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.
See 'git help git' for an overview of the system.

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git help init
```

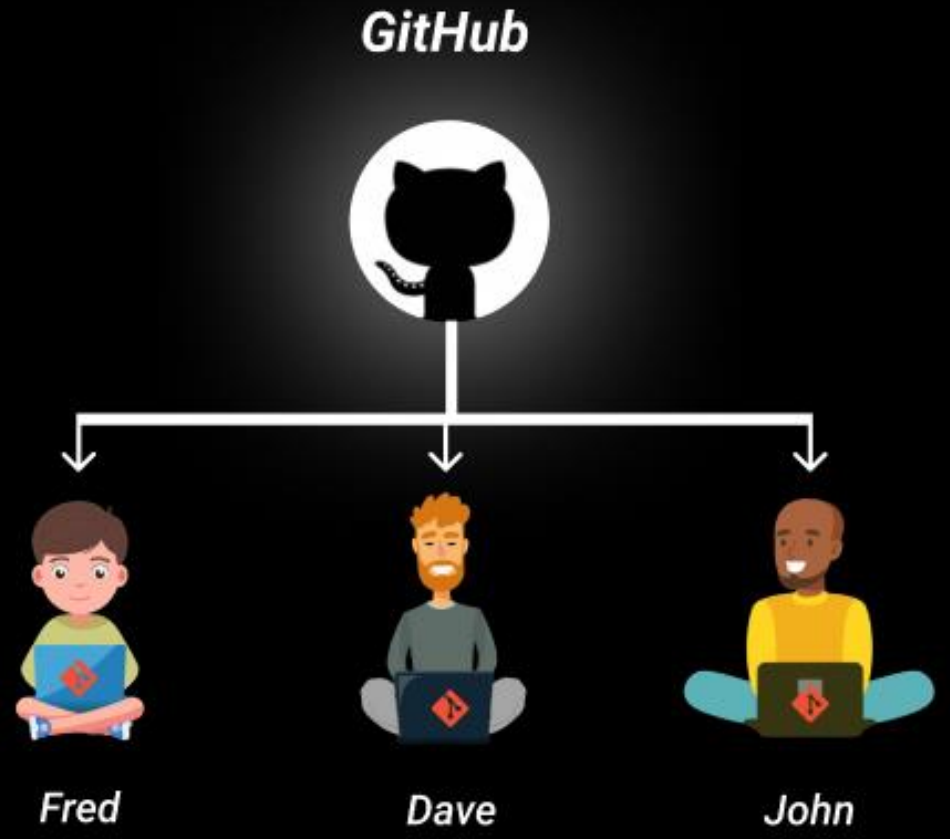
Git

What is Git?

Git is a distributed version control system source code management (SCM) system that records changes to a file or set of files over time, so that you can recall specific versions later.

It allows you to revert selected files back to a previous state, compare changes over time, see who last modified something that might be causing a problem, and more.

Git has a remote repository which is stored in a server and a local repository which is stored in the computer of each developer. This means that the code is not just stored in a central server, but the full copy of the code is present in all the developers' computers



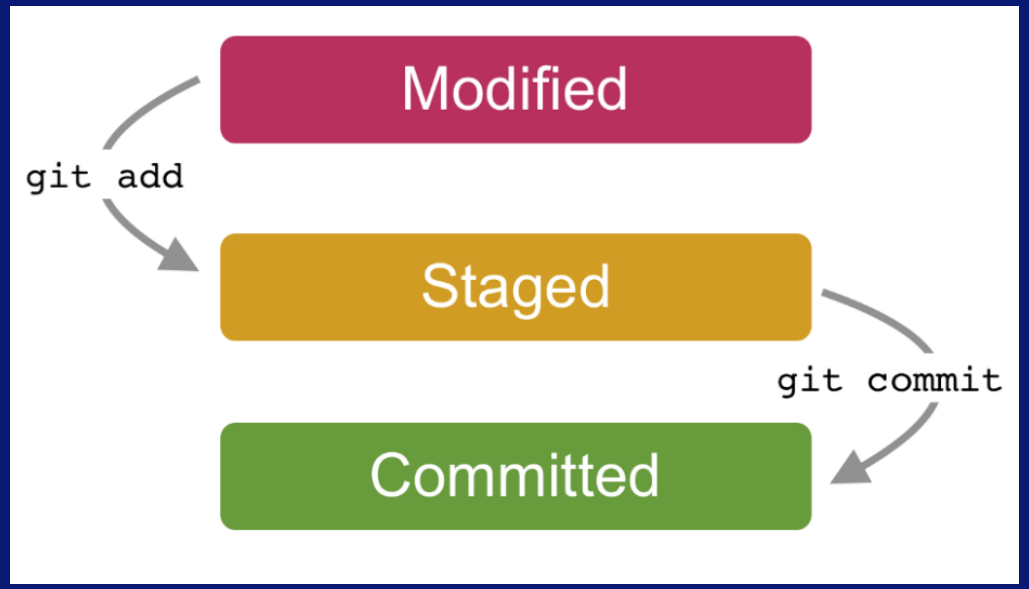
Git

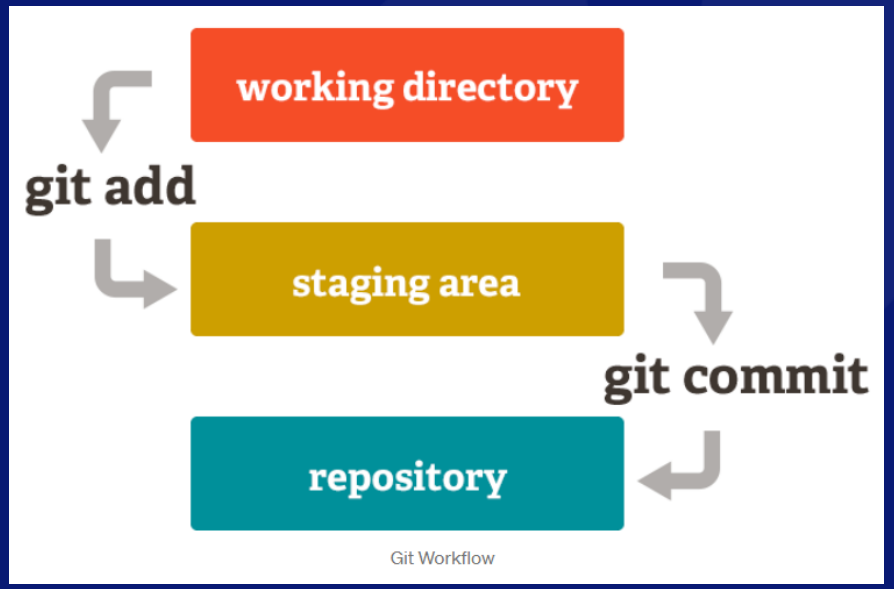
The 3 States of Git

Files in a repository go through three stages before being under version control with git

1. Modified means that you have changed the file but have not committed it to your database yet
2. Staged means that you have marked a modified file in its current version to go into your next commit snapshot
3. Committed means that the data is safely stored in your local database

- In addition to these three states, Git files live in one of three areas: the Working directory, Staging area, and the Git directory (your local repository)





Git

Create a local git repository

Create a project

- Create a new project folder in your local filesystem
- Go into this newly created project folder and add a local Git repository to the project using the following commands
  `git init`
- Now this project can be managed using Git
- Git creates a hidden folder `.git` in the project whenever you do `git init`
- The `.git` folder contains all the information that is necessary for your project in version control and all the information about commits, branches, remote repository address, etc. All of them are present in this folder. It also contains a log that stores your commit history so that you can roll back to history
- Without .git, the project is considered a local project and not a git project, that means you cannot perform any git operations

```
428991@LINL190904638 MINGW64 /d
$ mkdir git-demo

428991@LINL190904638 MINGW64 /d
$ cd git-demo

428991@LINL190904638 MINGW64 /d/git-demo
$ git init
Initialized empty Git repository in D:/git-demo/.git/

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -al
total 8
drwxr-xr-x 1 428991 1049089 0 Feb 10 19:55 ./
drwxr-xr-x 1 428991 1049089 0 Feb 10 19:55 ../
drwxr-xr-x 1 428991 1049089 0 Feb 10 19:55 .git/

428991@LINL190904638 MINGW64 /d/git-demo (master)
$
```

http://gitready.com/

Git

Create a local git repository

Add files to working area

- A newly added file will always be created in the working area and is untracked by default
- Now add some files to the local repository
- Check the status of the files using `git status`
- The status shows that `file1.txt` is an `untracked file` as it is not added to the staging area or Git has no idea what to do with this file yet
- `No commits yet` indicates there are no commits to the local repository yet
- The default Git branch is `master`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ touch file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My First File" > file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -al
drwxr-xr-x 1 428991 1049089  0 Feb 10 19:55 .git/
-rw-r--r-- 1 428991 1049089 14 Feb 10 19:57 file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

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

Git

Create a local git repository

Add files to staging/index using 'add'

- Unlike many version control systems, Git has a staging area (index)
- The Staging area is there to keep track of all the files which are to be committed. Any file which is not added to the staging area will not be committed. This gives the developer control over which files need to be committed at once
- git add lets you add files to the staging area
  - `git add file1.txt`
  - `git status`
- The status shows that `file1.txt` is ready to be committed to the local repository
- In case you want to add multiple files you can use:
  - `git add file1 file2 file3`
  - `git add .` or `git add -A`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git add file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt


428991@LINL190904638 MINGW64 /d/git-demo (master)
$
```

Git

Create a local git repository

Discard unstaged changes

- Before adding the files to staging, you can also discard the changes done
- Lets add some content to file1.txt and see the file status using `git status`
- The file is now in untracked state
- Git shows few hints whether to commit these changes or to discard them
- To commit simple proceed with `git add` or run `git restore <file-name>` to discard the changes
- Run `git status` again to see the status

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file1.txt
My First File

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My First Modification to file1" > file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file1.txt
My First Modification to file1

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git restore file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file1.txt
My First File

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

unstageg/untracked files can be committed in a new branch by switching to that branch

Git

Create a local git repository

Add files to staging

- Add few more files as shown in the figure and add to staging area
  - `git add file2.txt file3.txt` (or)
  - `git add .`
- Once a file is in the Staging area, you can unstage the file / untrack it using the `git rm --cached <file_name>` command

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ touch file2.txt file3.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My Second File" > file2.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My Third File" > file3.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

Untracked files:
  (use "git add <file>..." to include in what will
be committed)
        file2.txt
        file3.txt


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git add .

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt
        new file:   file3.txt
```

Git

Create a local git repository

Unstage files

- `git reset <filename>, git reset HEAD <filename>, git rm --cached <filename>, git restore --staged <filename>` all does the same work

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My First Modification to file1" > file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git add .

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git reset HEAD
Unstaged changes after reset:
M       file1.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

Git

Git Config

- The 'Git config' is a great method to configure your choice for the Git installation. Using this command, you can describe the repository behaviour, preferences, and user information
- The global config resides at `~/.gitconfig` in the user's home directory, while the repository specific config at `.git/config` inside the project
- Use `git config --help` for more info
- Before we commit any changes to the repo, we need to set the user name and email address for the git user so as to make Git aware of who has made the commits
- Run below commands to set the configuration
- `git config --global user.name "your name goes here"`
- `git config --global user.email "your email goes here"`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git config --global user.name "vikram"

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git config --global user.email "vikram.k@gmail.com"

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git config --global --list
http.sslverify=false
filter.lfs.required=true
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
user.name=vikram
user.email=vikram.k@gmail.com
core.editor="C:/Program Files (x86)/GitExtensions/GitExtensions.exe" fileeditor


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git config --list
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -
nosession -noPlugin
credential.helper=manager
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
pull.rebase=false
http.sslverify=false
filter.lfs.required=true
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
user.password=Virender418@
user.email=vikram.k@gmail.com
core.editor="C:/Program Files (x86)/GitExtensions/GitExtensions.exe" fileeditor
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
```

Git

Create a local git repository

Committing the code

- Committing is the process in which the code is added to the local repository
- Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you've done and why
- To give commit message, use -m flag
  - `git commit -m "Initial Commit"`
- If you don't use -m, Git will bring up an editor for you to write the commit message
- Each time you commit changes to the repo, Git creates a new SHA(20 byte number) that describes that state

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git commit -m "Initial Commit"
[master (root-commit) 35b408c] Initial Commit
 Committer: vikram <vikram.k@gmail.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
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:

    git commit --amend --reset-author

 3 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$
```

Git

Create a local git repository

View commit history

- `git log` command lists the commits made in that repository in reverse chronological order
- From the logs you can see the commit SHA ID, Author, Date and Commit message
- The **HEAD** points out the last commit in the current checkout branch. When you switch branches with `git checkout`, the HEAD is transferred to the new branch

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log
commit 53b694153e68bbed93e9a6cfc0c94d97c759b828 (HEAD -> master)
Author: 428991 <vikram.k@gmail.com>
Date:   Wed Feb 10 23:50:22 2021 +0530

    Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$
```

Git

Create a local git repository

Add few more files to the staging

- `git log`
- `git show head` is used to check the status of the Head

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log
commit 2fedc59b1199dfb2532903c7e378112f79dc714e (HEAD -> master)
Author: 428991
Date:   Thu Feb 11 08:16:27 2021 +0530

    Second Commit, Added file4.txt and file5.txt

commit 53b694153e68bbed93e9a6cfc0c94d97c759b828
Author: 428991
Date:   Wed Feb 10 23:50:22 2021 +0530

    Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git show head
commit 2fedc59b1199dfb2532903c7e378112f79dc714e (HEAD -> master)
Author: 428991 <vikram.babu-kunchala@alstomgroup.com>
Date:   Thu Feb 11 08:16:27 2021 +0530

    Second Commit, Added file4.txt and file5.txt

diff --git a/file4.txt b/file4.txt
new file mode 100644
index 0000000..6cafdcb
--- /dev/null
+++ b/file4.txt
@@ -0,0 +1 @@
+My Fourth File
diff --git a/file5.txt b/file5.txt
new file mode 100644
index 0000000..e5d949e
--- /dev/null
+++ b/file5.txt
@@ -0,0 +1 @@
+My Fifth File
```

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ touch file4.txt file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My Fourth File" > file4.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "My Fifth File" > file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file4.txt
        file5.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git add .

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file4.txt
        new file:   file5.txt


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git commit -m "Second Commit, Added file4.txt and file5.txt"
[master 2fedc59] Second Commit, Added file4.txt and file5.txt
 Committer: 428991 <vikram.babu-kunchala@alstomgroup.com>
Your name and email
on your username and hostname. Please check that they are accurate.
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:

    git commit --amend --reset-author

 2 files changed, 2 insertions(+)
 create mode 100644 file4.txt
 create mode 100644 file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Git

Create a local git repository

View commit history

- `git log --patch -2` shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries
- `git log --stat` shows abbreviated stats for each commit
- `git log --pretty=oneline` (or) `git log --oneline` --pretty changes the log output to formats other than the default. `--oneline` value for this option prints each commit on a single line
- `git log --name-only` – see files changed
- `git log -p :` If you also want to see complete diffs at each step
- `git log --stat --summary:` overview of the change

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --stat
commit 2fedc59b1199dfb2532903c7e378112f79dc714e (HEAD -> master)
Author: 428991 <vikram.k@gmail.com>
Date:   Thu Feb 11 08:16:27 2021 +0530

    Second Commit, Added file4.txt and file5.txt

 file4.txt | 1 +
 file5.txt | 1 +
 2 files changed, 2 insertions(+)

commit 53b694153e68bbed93e9a6cfc0c94d97c759b828
Author: 428991 <vikram.k@gmail.com>
Date:   Wed Feb 10 23:50:22 2021 +0530

    Initial Commit

 file1.txt | 1 +
 file2.txt | 1 +
 file3.txt | 1 +
 3 files changed, 3 insertions(+)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --pretty=oneline
2fedc59b1199dfb2532903c7e378112f79dc714e (HEAD -> master) Second Commit,
Added file4.txt and file5.txt
53b694153e68bbed93e9a6cfc0c94d97c759b828 Initial Commit
```

Git

Git ignore (.gitignore)

- Git sees every file in your working copy as one of three things: tracked - a file which has been previously staged or committed; untracked - a file which has not been staged or committed; or ignored - a file which Git has been explicitly told to ignore
- Git is meant to store only source files and never the generated files or the personal config files with confidential information
- Any generated code/binaries should not be committed due to following reasons:
  - The generated files can be recreated at any time on the client side and may need to be created in a different form, so It is a waste of time and space to store them in git
  - The generated files are often large than the original source files themselves. Putting them in the repo means that everyone needs to download and store those generated files, even if they're not using them. This increased the clone times
  - Git usually stores only changes made to the file in the subsequent commits by using diff algorithm. Binary files don't really have good diff tools, so Git will frequently just need to store the entire file each time it is committed
- Any file added to .gitignore file in the root project directory will not show up as untracked file while doing git status and so can be ignored. Common files to be ignored include:
  - dependency caches, such as the contents of /node_modules or /packages
  - compiled code, such as .o, .pyc, and .class files
  - build output directories, such as /bin, /out, or /target
  - files generated at runtime, such as .log, .lock, or .tmp
  - hidden system files, such as .DS_Store or Thumbs.db
  - personal IDE config files, such as .idea/workspace.xml

Git Ignore patterns:
https://www.atlassian.com/git/tutorials/saving-changes/gitignore
https://github.com/github/gitignore

Git

Git ignore (.gitignore)

- Let's create a file called TODO.txt which has todo list for personal reference
- Add some content to this file and check the status
- Git shows the file as untracked. But these are personal preferences and should not be committed or bring under Git's radar
- We can add this file to `.gitignore` file to instruct git to ignore the changes done to TODO file and not to track
- We should commit `.gitignore` file so others can use it and updated it with their own to be ignored content like files generated code, log files, cache files and IDE specific files

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
755a2fc (HEAD -> master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "[TODO]: Change the login button layout" > TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        TODO.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "TODO.txt" >> .gitignore

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat .gitignore
TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt
TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

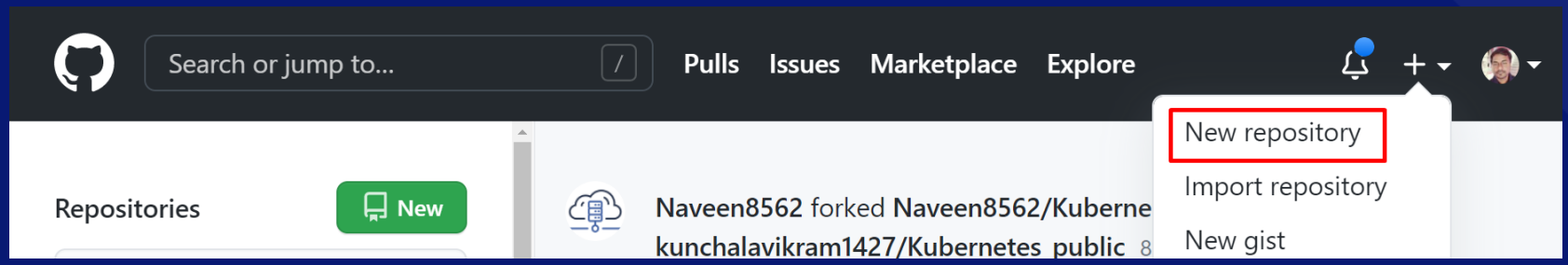
Git

Create a Remote Repo

Remote Repo in Github

- Files/Project which you have created earlier stays within your local repository
- If you want to share these files with other developers or team members to collaborate, you need to push the local repo to a remote repository like GitHub
- After you have a remote repository set up, you upload (push) your files and revision history to it. After someone else makes changes to a remote repo, you can download (pull) their changes into your local repo

- Go to https://github.com/ and create a New Repository

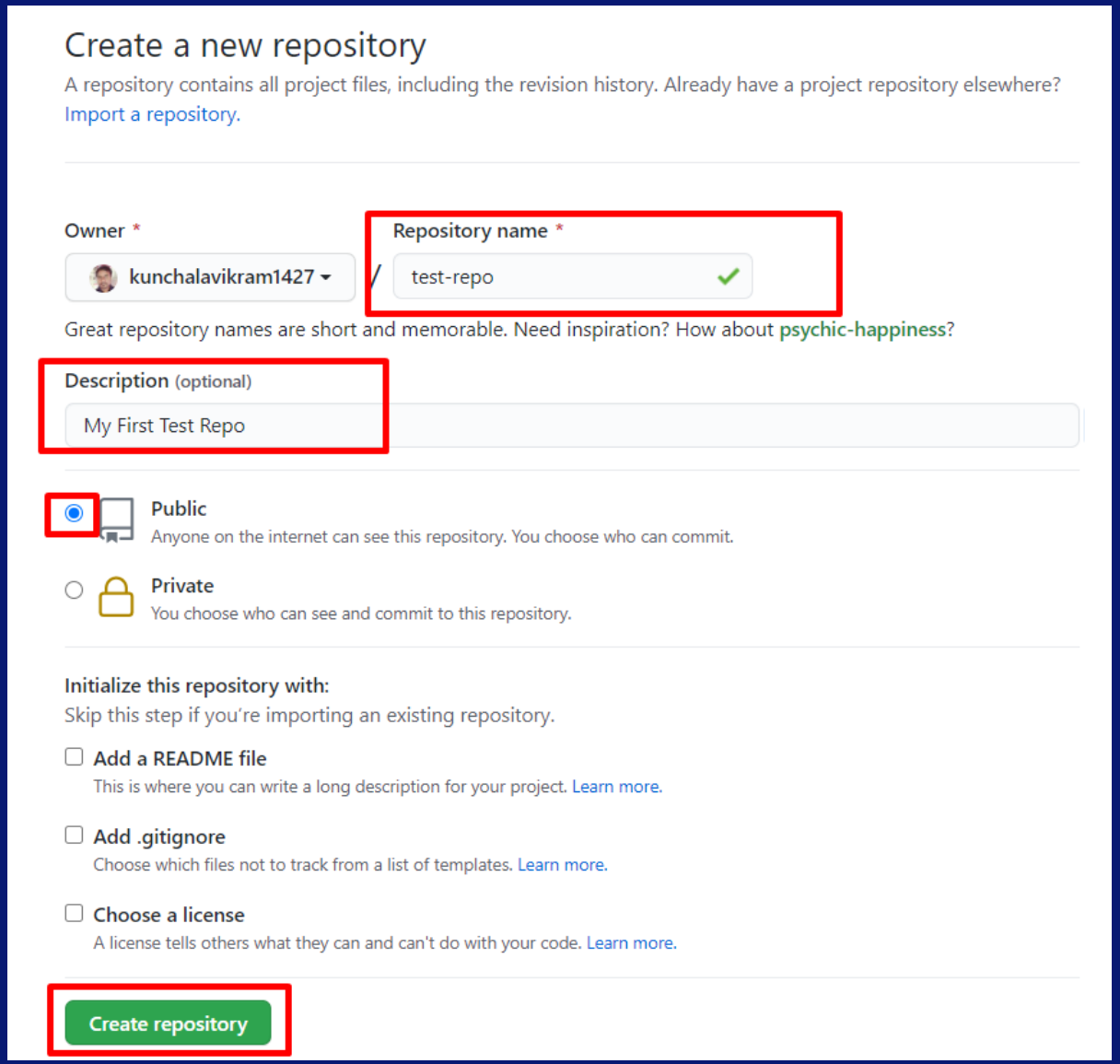


Git

Create a Remote Repo

Remote Repo in Github

- Give a Repository name, Description and Make the repo public
- Click on Create Repository to create it

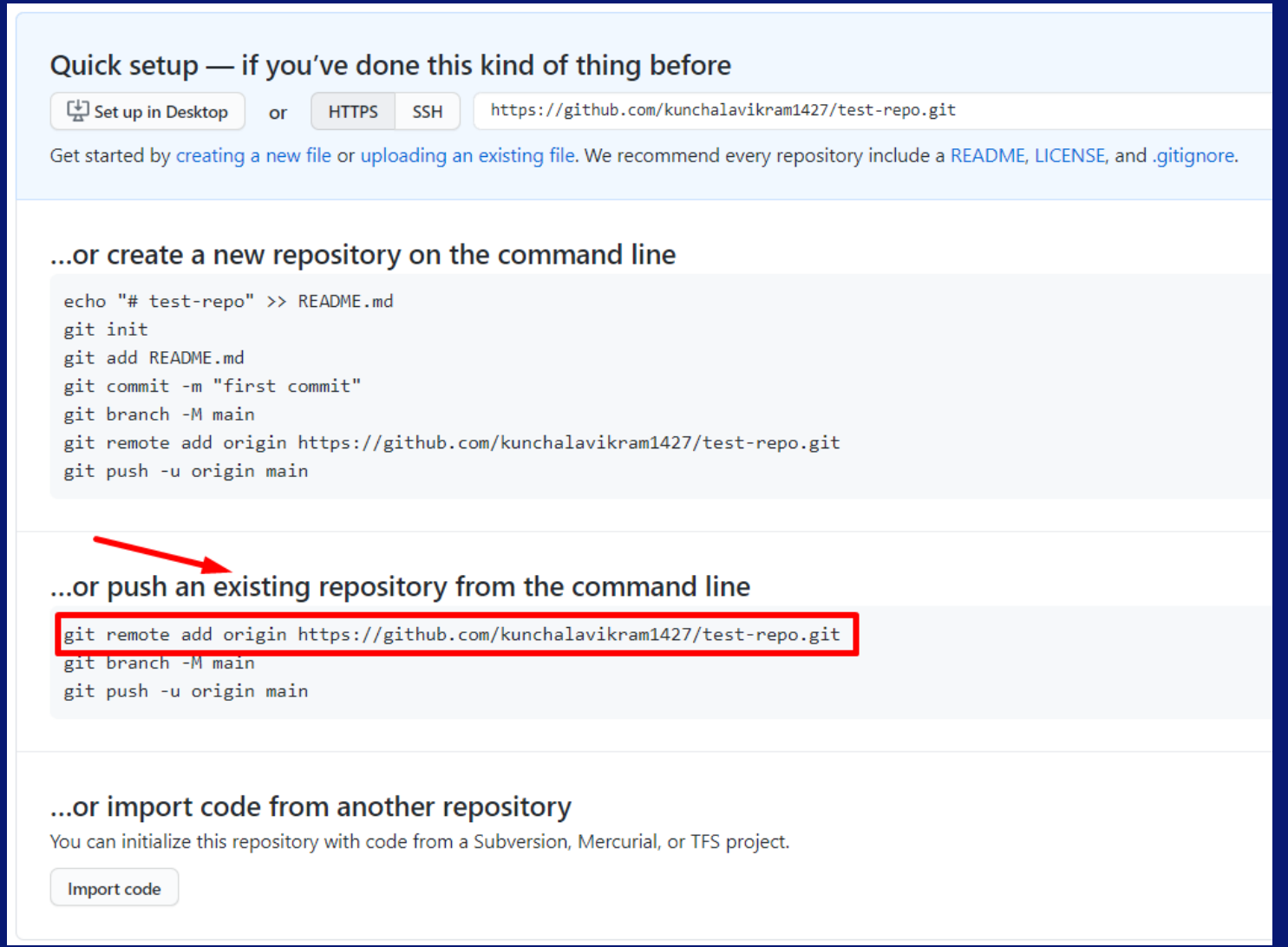


Git

Create a Remote Repo

Remote Repo in Github

- Once created, GitHub will prompt to create a new repo or to add a repo we have created locally
- In our case, since we've already created a local repository, we can push it directly
- Git will also show us the instructions to be followed to push our local repo.
- In this case we need to push to the URL that is shown

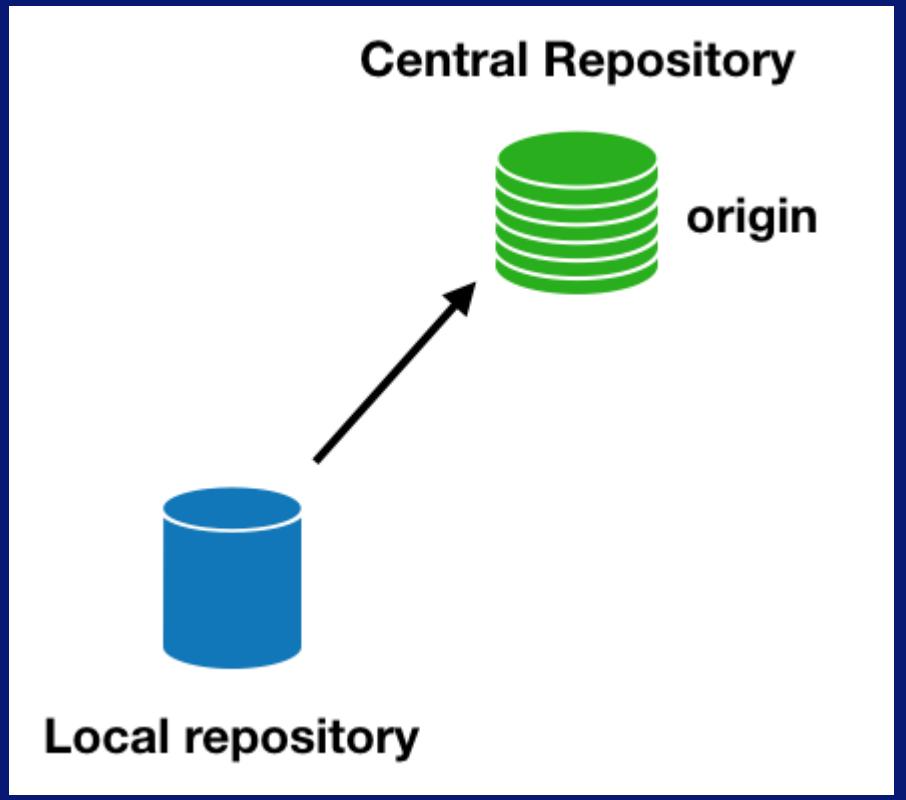


Git

Create a Remote Repo

What Is Origin?

- `Origin` is an alias to the remote repository
- We assign an alias so we won't have to write out the URL of the remote repo every time we want to work with the remote repository
- While `origin` is the alias that most people prefer to use, it is not a standard and we can use our own naming conventions



Git

Remote Repo

Adding the remote repo

- Add the remote repo by running
  - `git remote add origin https://github.com/<your-user-account>/test-repo.git`
- `remote` command adds a remote repository to the current repository
- Check the remote URLs and their aliases
  - `git remote -v`

- Remove the remote repo by running
  - `git remote rm origin`

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git remote add origin https://github.com/kunchalavikram1427/test-repo.git

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git remote -v
origin  https://github.com/kunchalavikram1427/test-repo.git (fetch)
origin  https://github.com/kunchalavikram1427/test-repo.git (push)

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git config --list
.
.
.
user.name=vikram
remote.origin.url=https://github.com/kunchalavikram1427/test-repo.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
```

Git

Remote Repo

Push the files to the origin

- `git push` pushes our local repo changes to the added remote repo. **origin** represents the remote repo and **master** is the branch you want to push to. Git will prompt for your git credentials during push
  - `git push -u origin master`

- We can omit origin and -u master parameters from the git push command with the following two Git configuration changes:
  $ git config remote.pushdefault origin
  $ git config push.default current
- The first setting saves you from typing origin every time. And with the second setting, Git assumes that the remote branch on the GitHub side will have the same name as your local branch.

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git push -u origin master
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (12/12), 854 bytes | 427.00 KiB/s, done.
Total 12 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/kunchalavikram1427/test-repo.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$
```

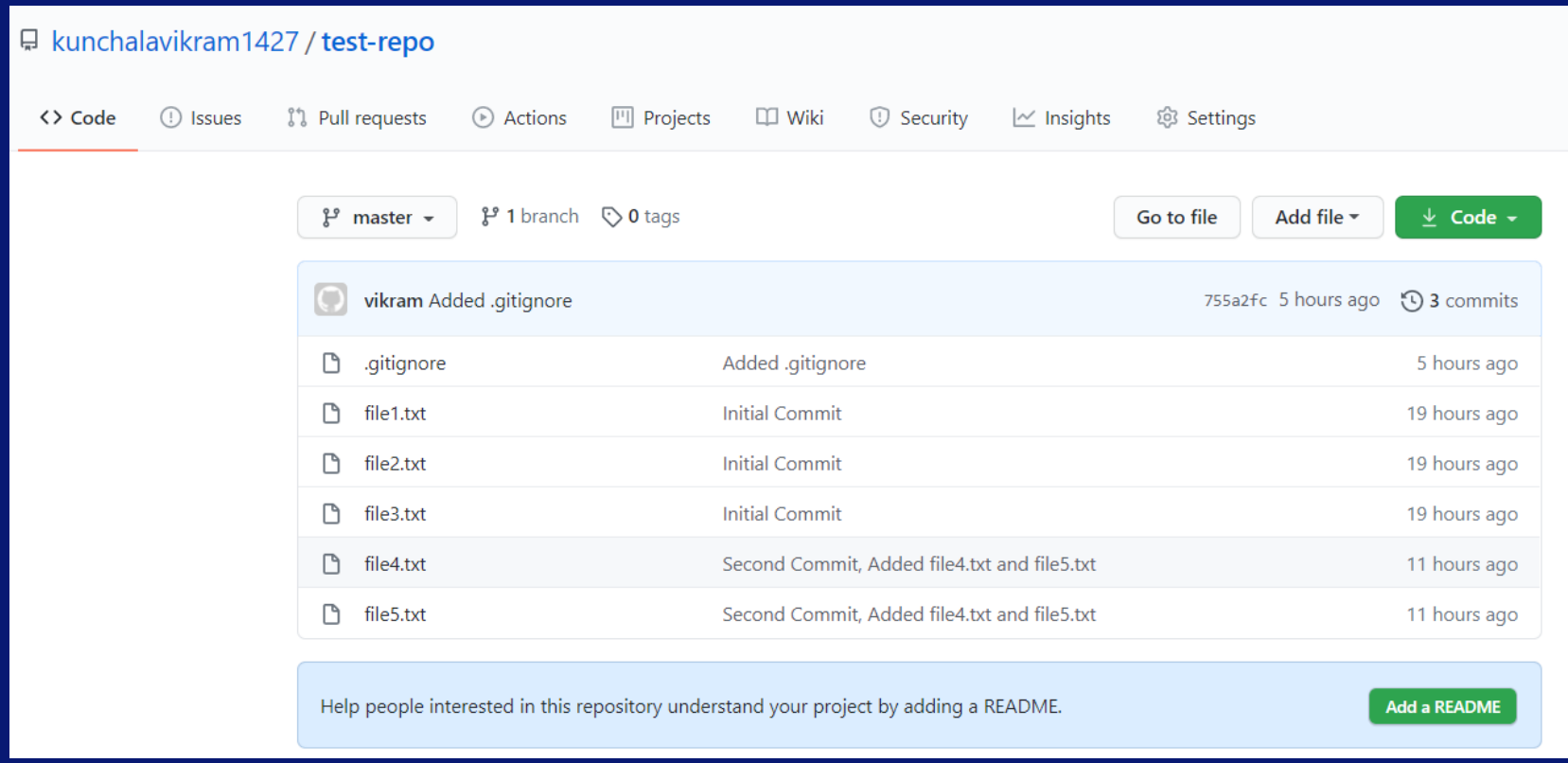
Git

git:kunchalavikram1427

Remote Repo

Push the files to the origin

- Go to your GitHub account and see the changes



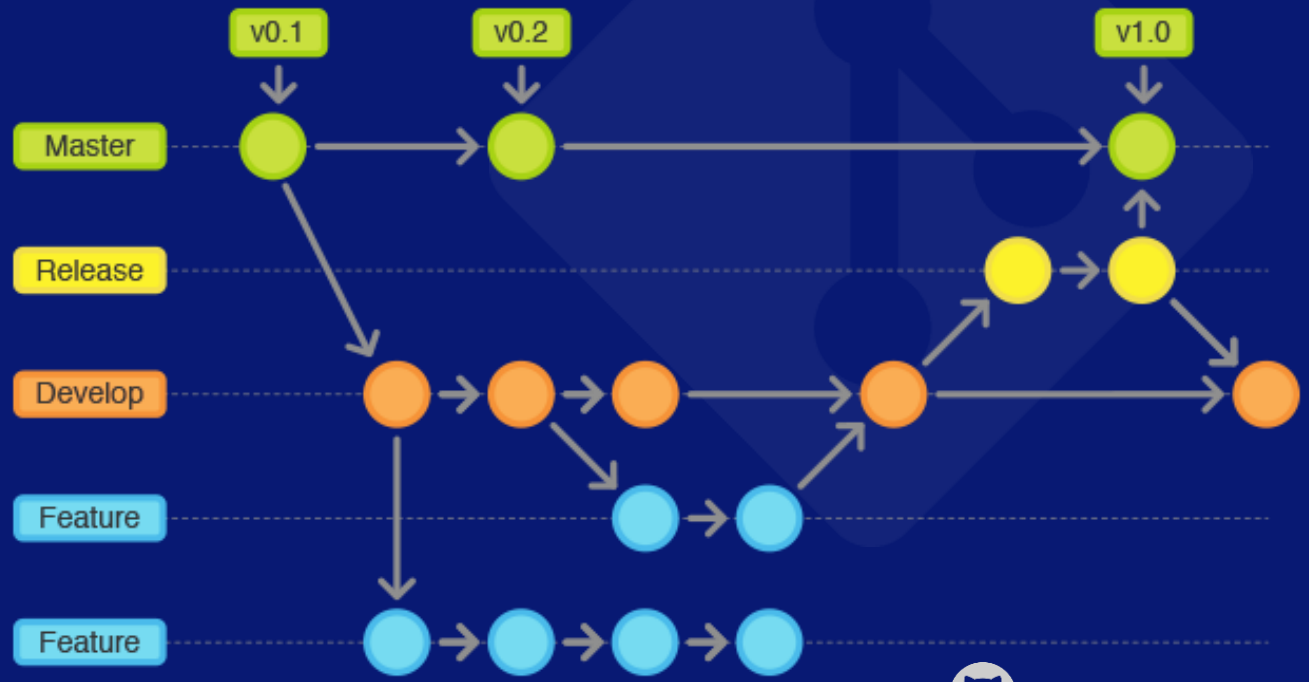
/kunchalavikram1427

Git

Branches

- GIT Branch is about maintaining the separate line of development. The default branch is ***Master***
- Git lets you branch out from the original code base. This lets you more easily work with other developers, and gives you a lot of flexibility in your workflow
- Let's say you need to work on a new feature for a website. You create a new branch and start working. You haven't finished your new feature, but you get a request to make a rush change that needs to go live on the site today. You switch back to the master branch, make the change, and push it live. Then you can switch back to your new feature branch and finish your work. When you're done, you merge the new feature branch into the master branch, and both the new feature and rush change are kept!

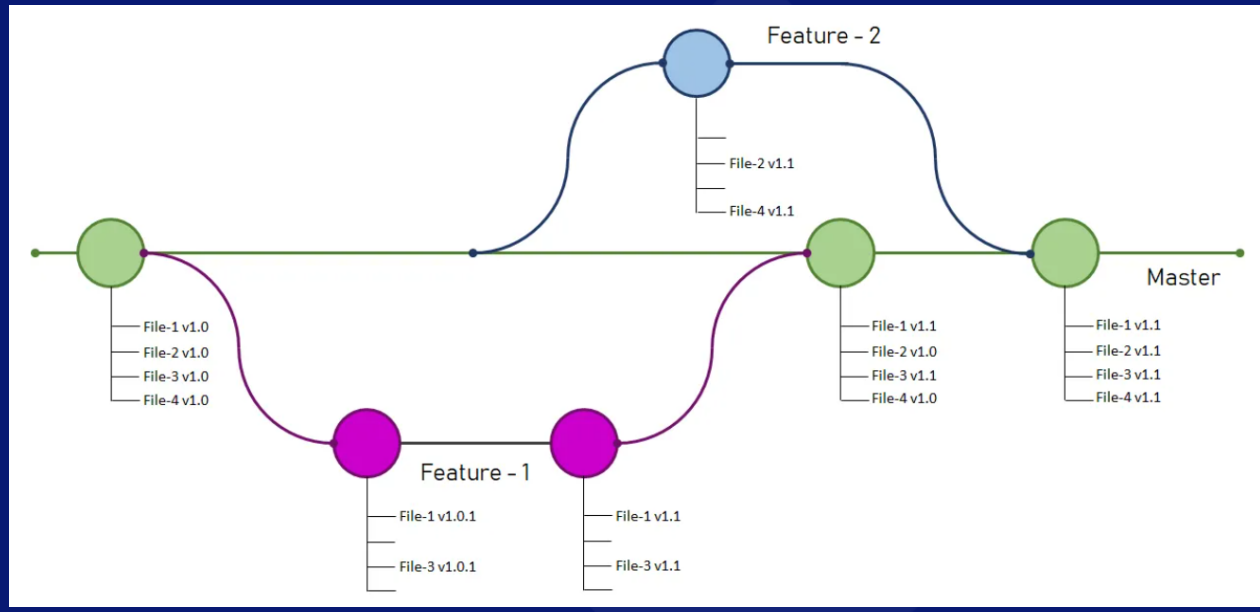
```
Git Branch is basically a
pointer to the last commit
```



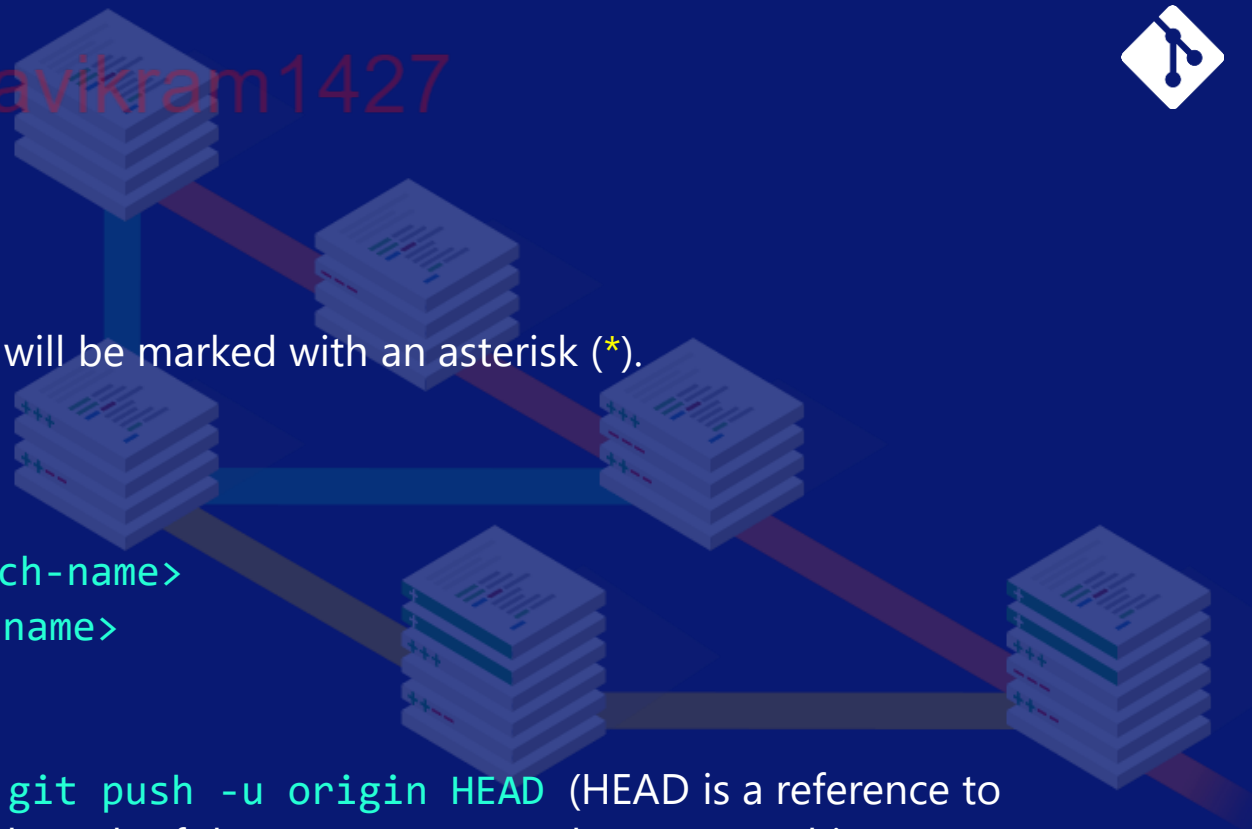
Git

Branches

- Basically, A branch is the isolated and independent line of developing the feature
- Assume the middle line as the master branch where the code is stable, working and updated
- Then, assume a developer-1 is assigned to develop Feature – 1. So, Developer-1 is cutting a branch from the master branch which is indicated in the diagram as in Magenta colour
- Similarly Developer-2 is assigned to develop Feature-2 and he is cutting a branch from the master branch which is indicated in Blue colour
- When both of them are done with their changes they merge their branches to the master and these feature branches can be safely deleted
- Let's assume the codebase is the set of four files called File-1, File-2, File-3, and File-4. So, when Developer-1 is taking the branch from master to develop feature-1 (assume File-1 and File-3 are going to be changed for the development). So, the developer can safely commit his changes on both files into the master branch. Vice versa, When Developer-2 is taking the branch from master to develop feature-2 (assume File-2 and File-4 are going to be changed for the development).



Git



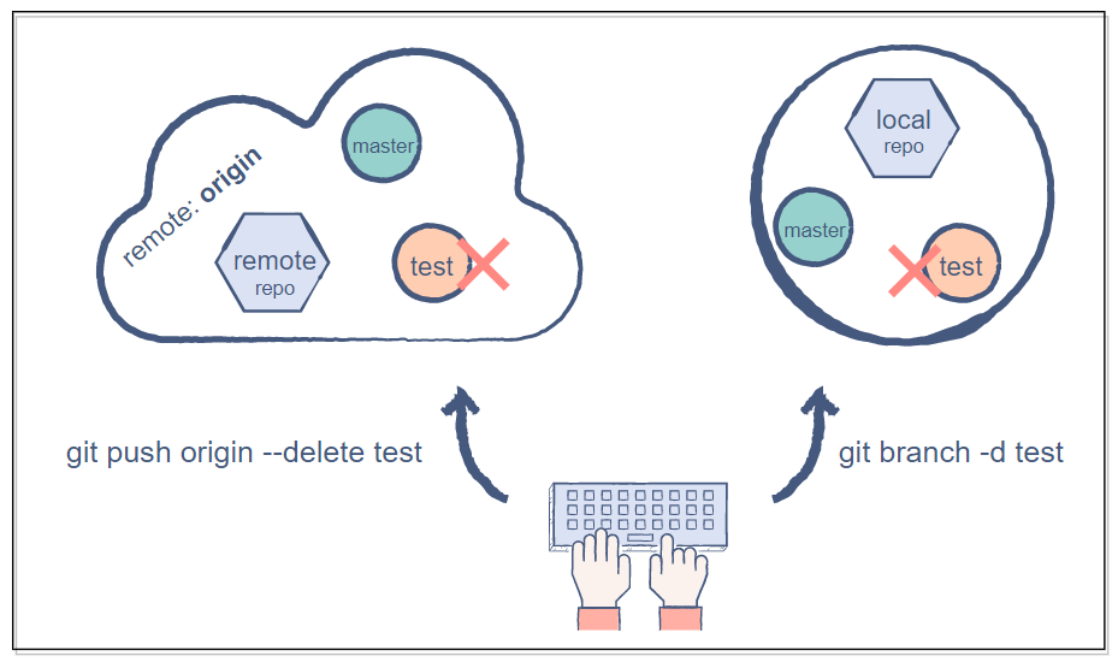
Branch commands

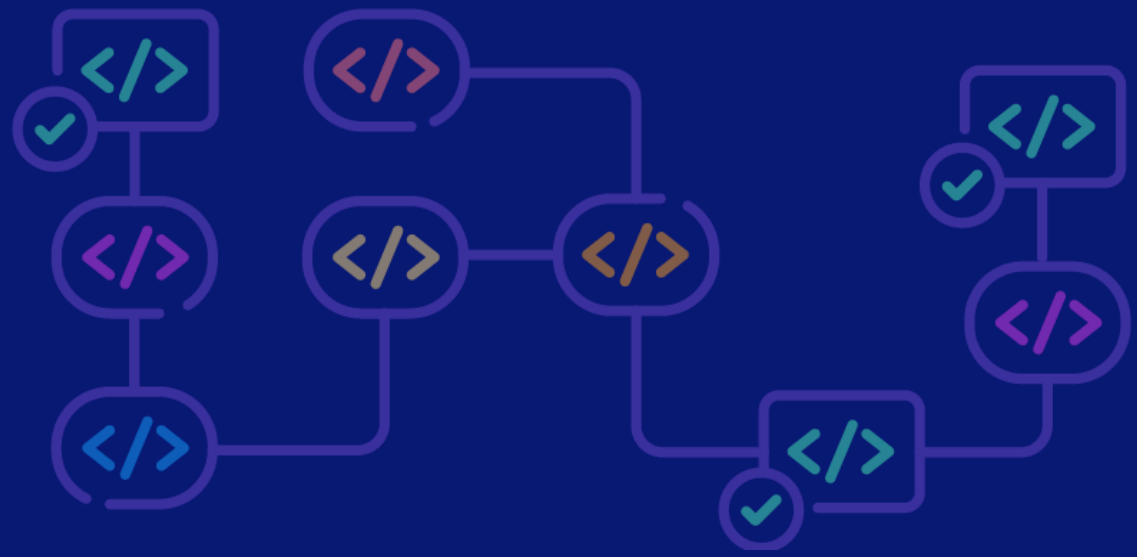
- Check What Branch You're On: `git status`
- To see local branches: `git branch` (The current local branch will be marked with an asterisk (*).
- To see remote branches: `git branch -r`
- To see all local and remote branches: `git branch -a`
- Create a New Branch: `git branch <branch-name>`
- Switch to a Branch In Your Local Repo: `git checkout <branch-name>`
- Create a branch and checkout: `git checkout -b <branch-name>`
- Rename a Branch: `git branch -m <branch-name>`
- To get a list of all branches from the remote: `git pull`
- Push to a Branch: `git push -u origin <branch-name>` or `git push -u origin HEAD` (HEAD is a reference to the top of the current branch, so it's an easy way to push to a branch of the same name on the remote. This saves you from having to type out the exact name of the branch!
- If your local branch already exists on the remote: `git push`
- You'll want to make sure your working tree is clean and see what branch you're on: `git status`
- `git merge <branch-name>` (If you want this branch to merge to master, you should already be in master branch before merging. If not run `git checkout master`)
- To delete a remote branch : `git push origin --delete <branch-name>`
- To delete a local branch: `git branch -d <branch-name>` (or) `git branch -D <branch-name>` (the -d option only deletes the branch if it has already been merged. The -D option is a shortcut for --delete --force, which deletes the branch irrespective of its merged status)

Git

Branch commands

- Delete a remote branch: `git push origin --delete :<branch-name>` or `git push origin :<branch_name>`
- Rename a local branch by being in the same branch: `git branch -m <new-branch-name >`
- Rename a local branch by being in another branch: `git branch -m <old-branch-name > <new-branch-name >`
- If you are in the branch which needs to be renamed in remote, then pass the following command with upstream argument -u: `git push origin -u new-name`



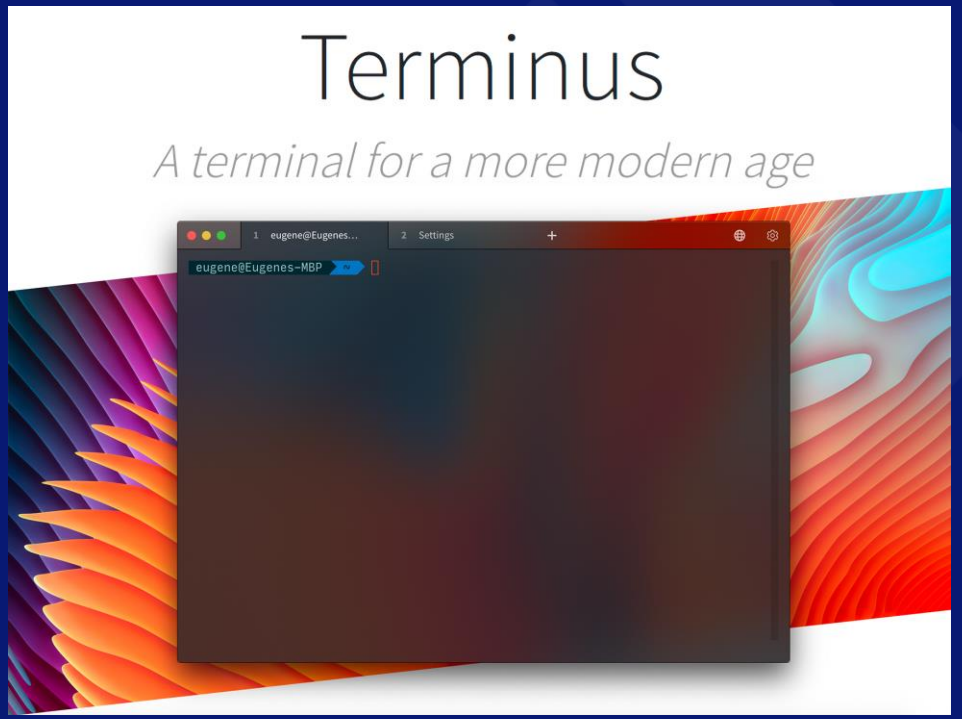


Git

CLI

- Git can be used using a command line (terminal), or a desktop app that has a GUI (graphical user interface) such as SourceTree shown next
- I prefer using terminus https://github.com/Eugeny/terminus which is a CLI tool. It directly shows the current git branch as seen below and has many features like Serial, SSH protocols
- Download it from here https://github.com/Eugeny/terminus/releases/tag/v1.0.132

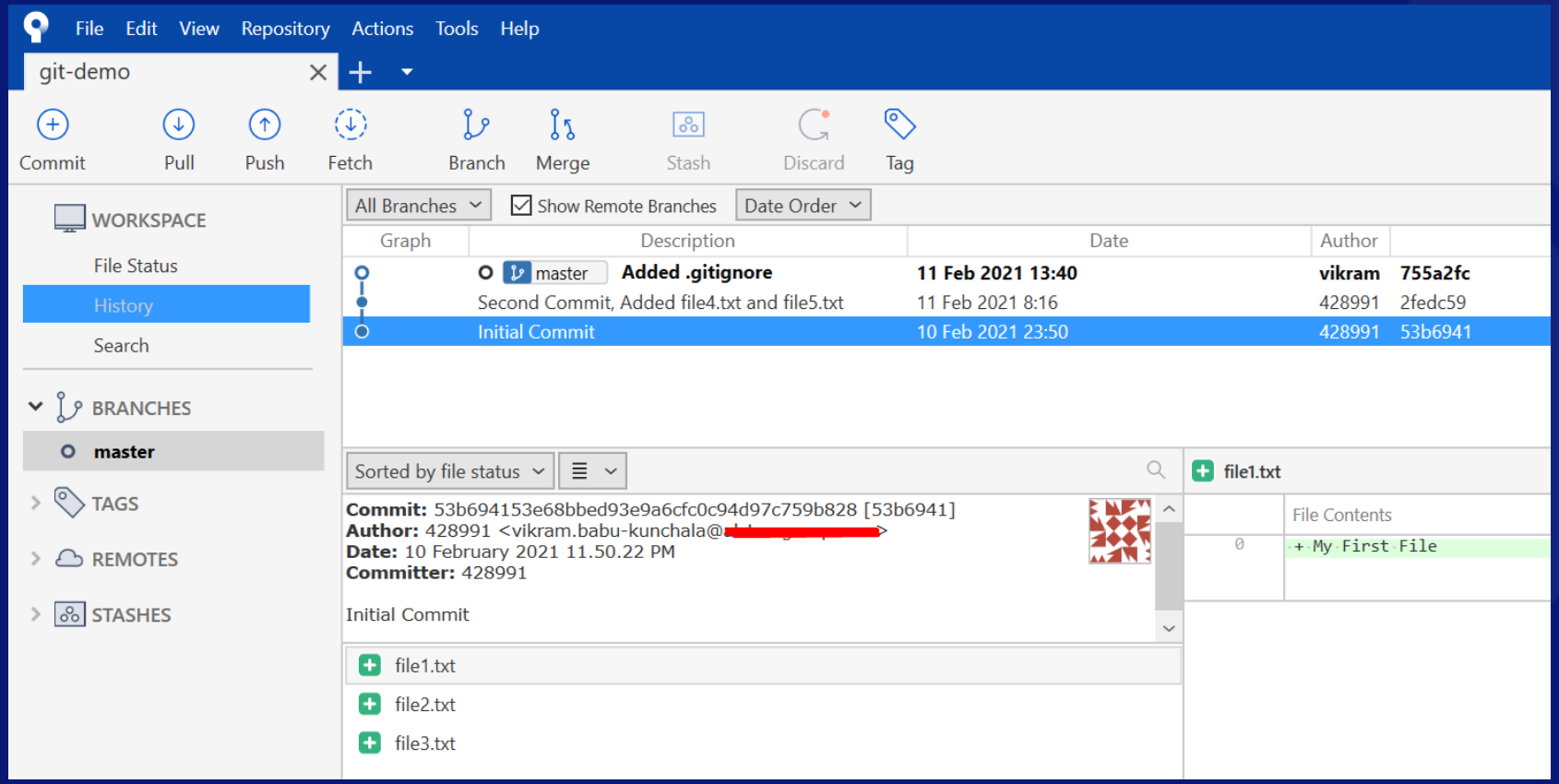
```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/git-demo (master)
$

428991@LINL190904638 MINGW64 /d/git-demo (master)
$
```



Git

SourceTree

- To visually see every commit and branches, download SourceTree from https://www.sourcetreeapp.com/
- During installation, SourceTree will automatically detect your Git installation
- Once installed, import the git project created or add a remote origin
- As you can see below, during our demo we have made 3 commits to the master branch



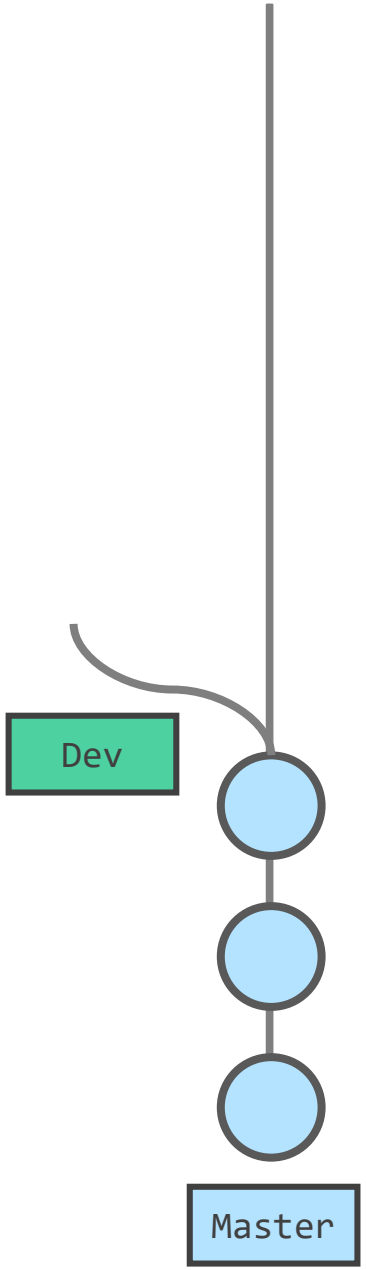
Git

Branches

- Show current branch. The current local branch will be marked with an asterisk (*)
  - `git status`
- List all of the branches in your repository
  - `git branch`
  - `git branch --list`
- Create a new branch dev
  - `git branch dev`
- Switch to another branch
  - `git checkout dev`
- Create and checkout at once
  - `git checkout -b dev`
- Check HEAD (HEAD points out the last commit in the current checkout branch)
  - `git show head`

For making commits to a new branch, you should be in that branch already!

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git log --oneline
755a2fc (HEAD -> dev, master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch
* master

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch dev

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch
  dev
* master

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git checkout dev
Switched to branch 'dev'

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git status
On branch dev
nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git branch
* dev
  master

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git show head
commit 755a2fc69ec0ef2b740cf5e2b8c8768dc69e20ec (HEAD -> dev, master)

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git log --oneline
755a2fc (HEAD -> dev, master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```



Git

Branches

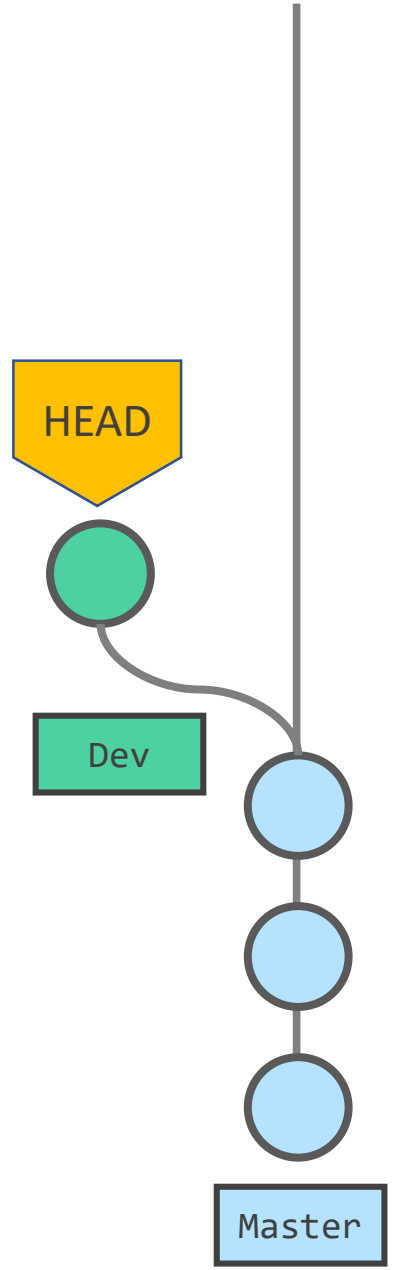
Push the files to origin

- Add a new file file6.txt ad check the status
  - `git status`
- Add the file to staging
  - `git add .`
- Commit to local
  - `git commit -m "Added Sixth File to Dev Branch"`
- Push to remote dev branch
  - `git push origin dev`
- Check log
  - `git log --oneline`

Add and commit in 1 shot:
`git commit -a -m "Commit message"`

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt
file5.txt  TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ touch file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ echo "My Sixth File" > file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git status
On branch dev
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file6.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git add .
warning: LF will be replaced by CRLF in file6.txt.
The file will have its original line endings in your working directory

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git commit -m "Added Sixth File to Dev Branch"
[dev 7f953dd] Added Sixth File to Dev Branch
 1 file changed, 1 insertion(+)
 create mode 100644 file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git push origin dev
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 295 bytes | 295.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'dev' on GitHub by visiting:
remote:      https://github.com/kunchalavikram1427/test-repo/pull/new/dev
remote
To https://github.com/kunchalavikram1427/test-repo.git
 * [new branch]      dev -> dev
```

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git log --oneline
7f953dd (HEAD -> dev, origin/dev) Added Sixth File to Dev Branch
755a2fc (origin/master, master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```



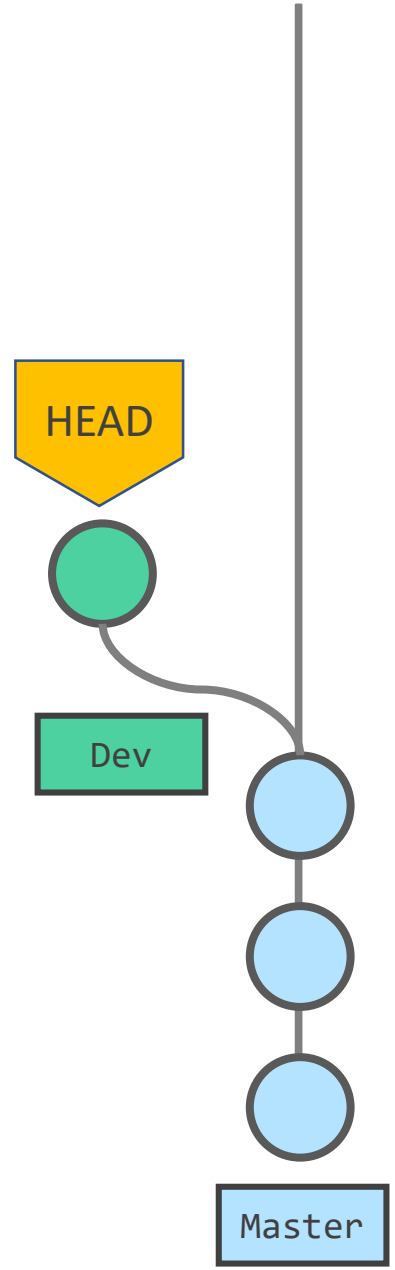
Git

Branches

Push the files to origin

- To know from which commit/branch, the **dev** branch is created, run the below command by being in the **dev** branch
  - `git log --graph --decorate`
- From the log it is evident that dev branch is created from the master
- Similarly, to know the branching information of a particular branch, switch to that branch and run the command

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git log --graph --decorate
* commit 7f953dd9292ca608e7a803fda7bfc682f7d01070 (HEAD -> dev, origin/dev)
| Author: vikram
| Date:   Thu Feb 11 19:10:49 2021 +0530
|
|     Added Sixth File to Dev Branch
|
* commit 755a2fc69ec0ef2b740cf5e2b8c8768dc69e20ec (origin/master, master)
| Author: vikram
| Date:   Thu Feb 11 13:40:26 2021 +0530
|
|     Added .gitignore
|
* commit 2fedc59b1199dfb2532903c7e378112f79dc714e
| Author: 428991
| Date:   Thu Feb 11 08:16:27 2021 +0530
|
|     Second Commit, Added file4.txt and file5.txt
|
* commit 53b694153e68bbed93e9a6cfc0c94d97c759b828
  Author: 428991
  Date:   Wed Feb 10 23:50:22 2021 +530

      Initial Commit
```

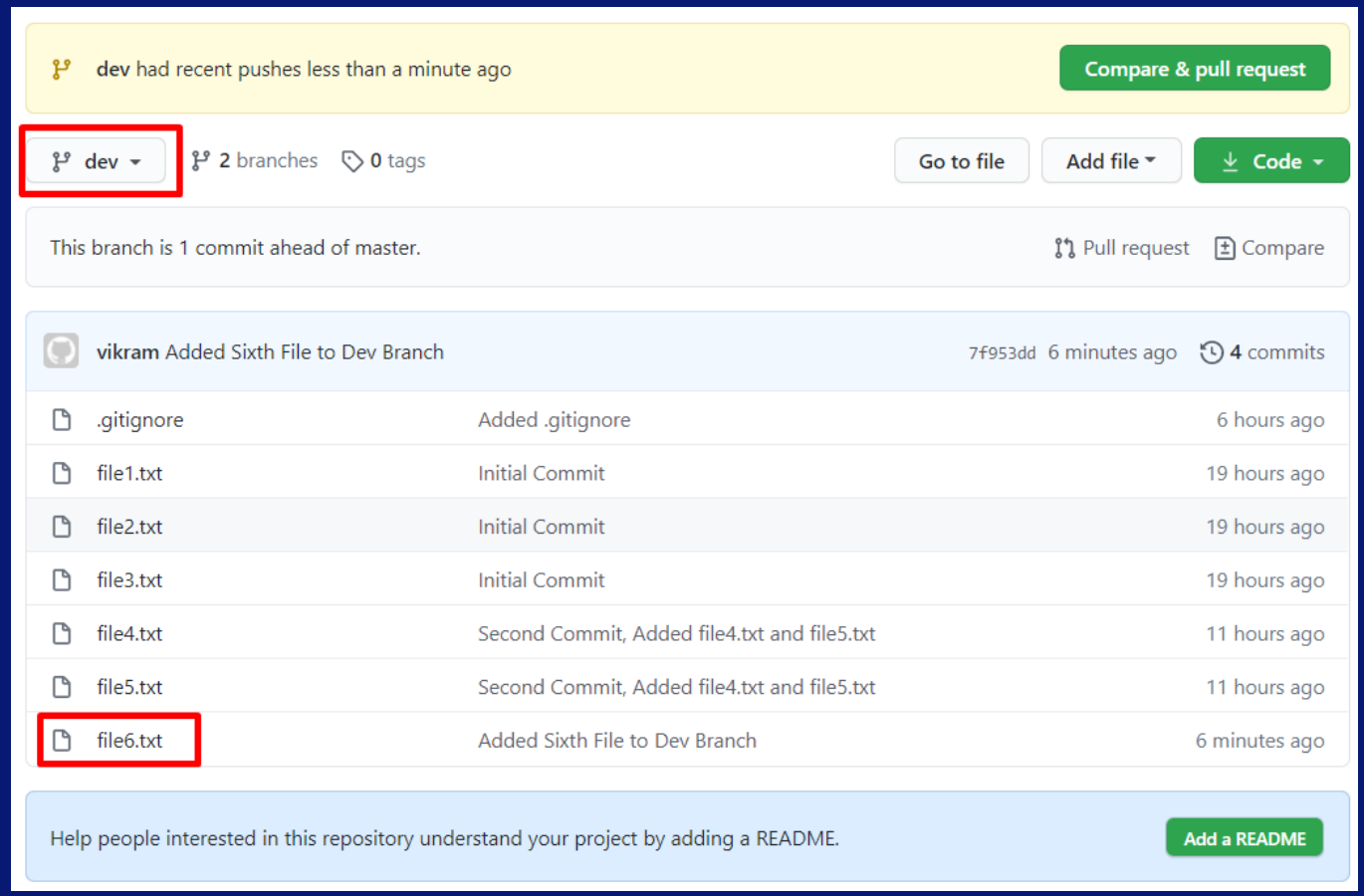


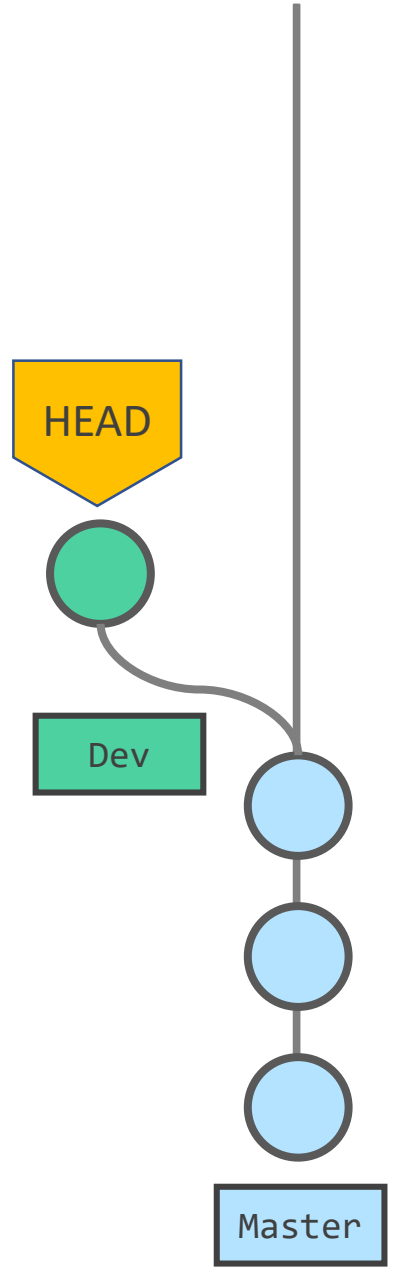
Git

Branches

Push the files to origin

- Check your files in the repository in dev branch



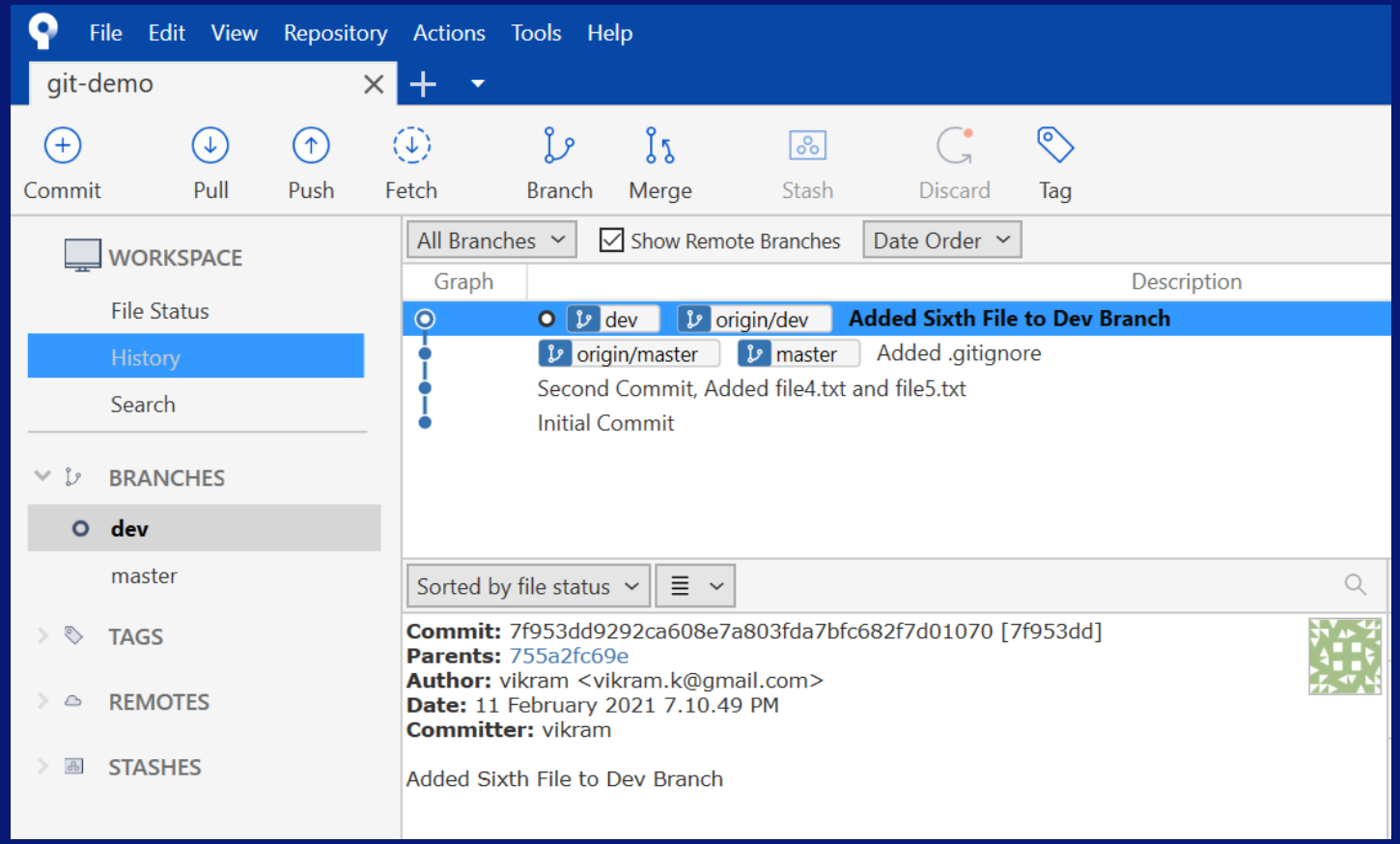


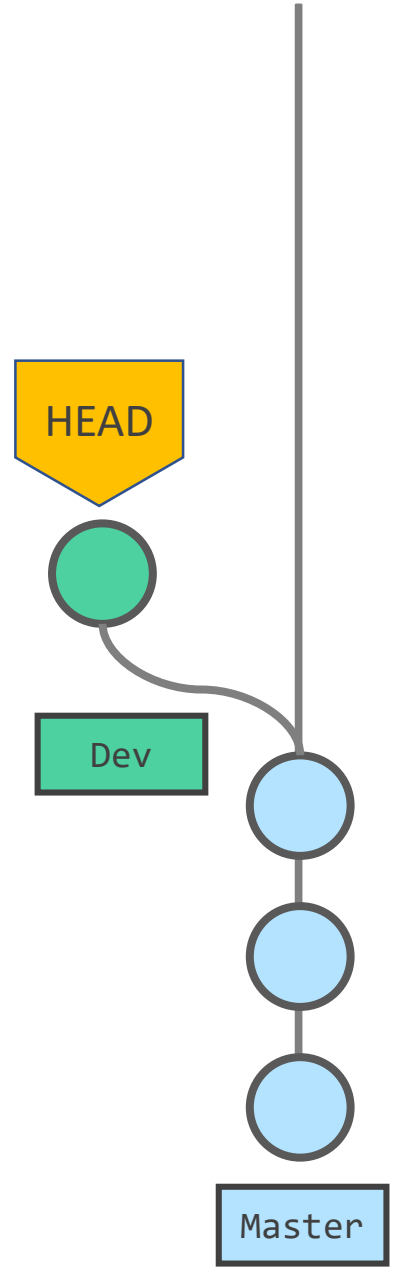
Git

Branches

Push the files to origin

- Verify the same in SourceTree





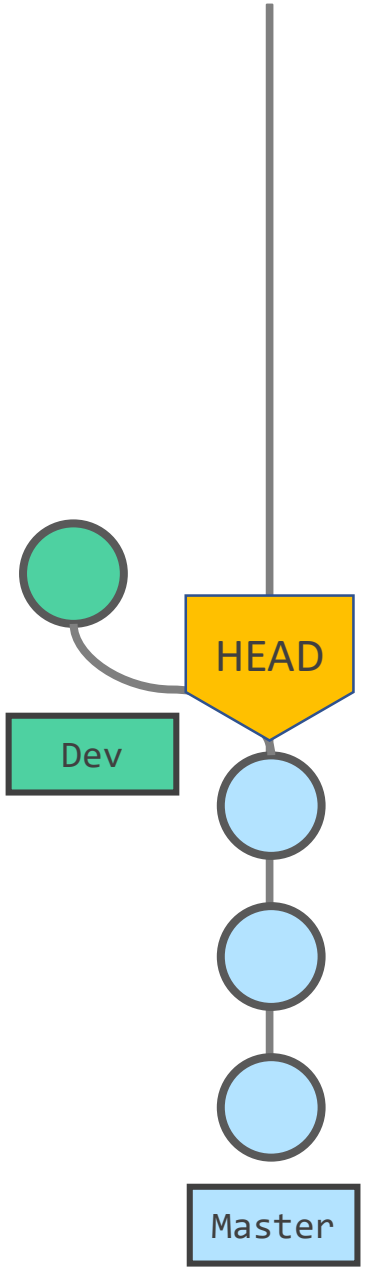
Git

Branches

Checkout

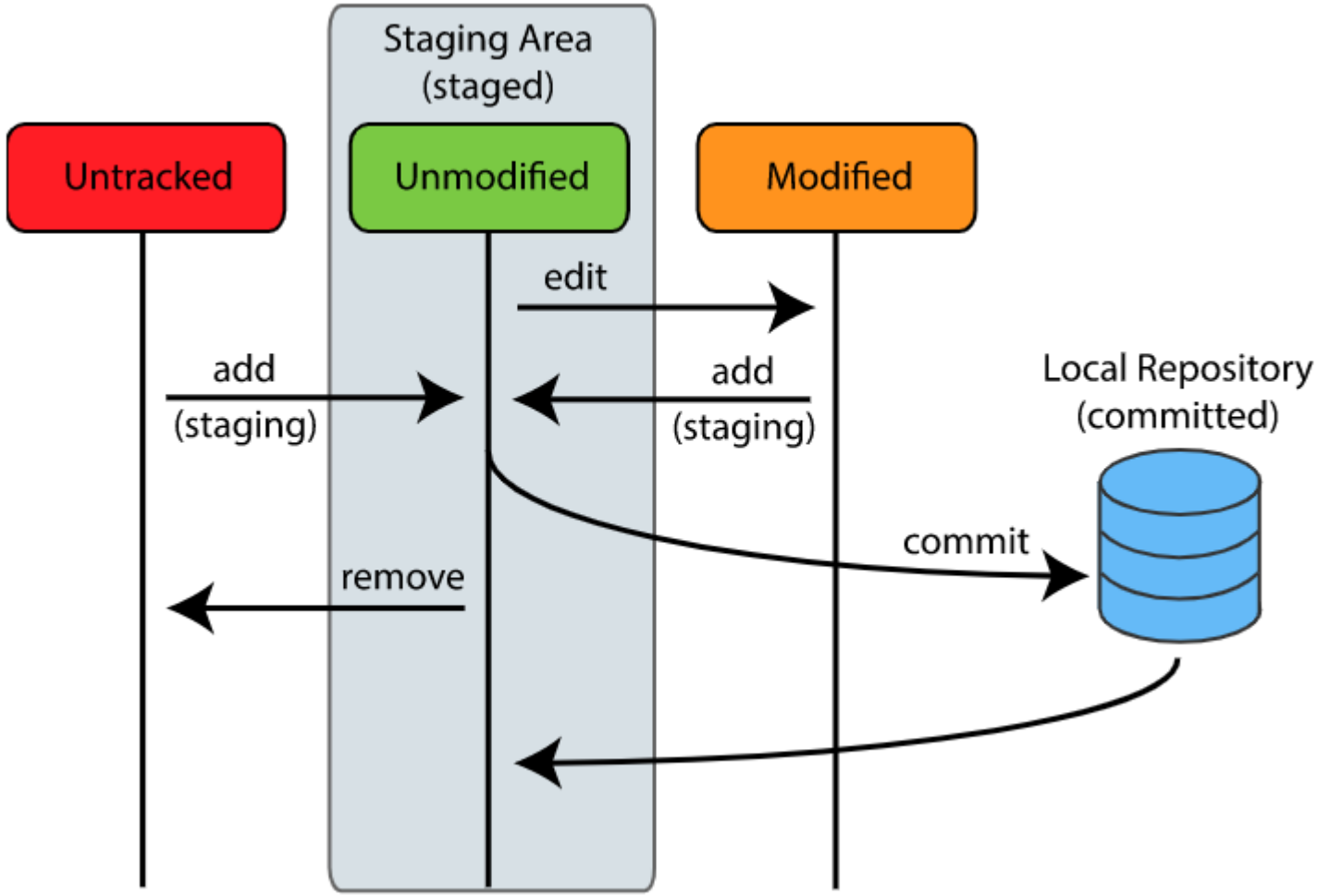
- The git checkout command operates upon three distinct entities: files, commits, and branches
- The git checkout command lets you navigate between the branches created by git branch
- Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch
  - `git checkout master` – switches the branch to master and update the files as well
- Since file6.txt is only in dev branch, the file is missing in the master branch. Master is 1 commit behind the dev branch

```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt  file6.txt  TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt  TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git show head
commit 755a2fc69ec0ef2b740cf5e2b8c8768dc69e20ec (HEAD -> master, origin/master)
```



Git

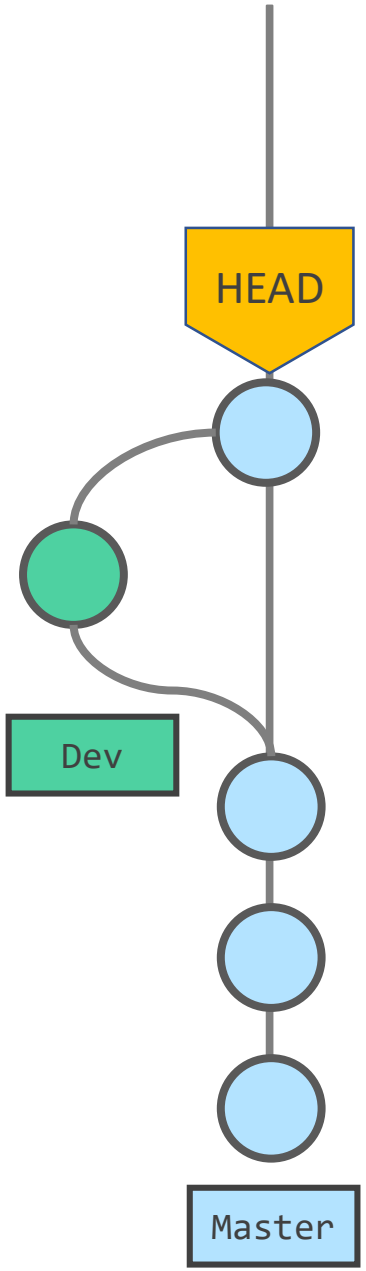
Workflow



Git

Merge

- A branch is nothing but a pointer to the latest commit in the Git repository
- Currently, dev branch is ahead of the master by 1 commit
- Let's merge all the changes in the dev branch to master using `git merge` command
- In order to merge the code from the dev branch into the master branch, we should first checkout to master
  - `git checkout master`
  - `git merge dev`
- After running these 2 commands, the merge should be successful. In this example, there are no conflicts. But in real projects, there will be conflicts when a merge is being done.
- We will see about Merge conflicts in the later part of the Tutorial
- Run `git log` now and you will notice that the master now has now 4 commits and the file6.txt which is created in dev branch
- Since dev branch is now merged to master, it can be safely removed
- To delete a local branch: `git branch -d <branch-name>` (or) `git branch -D <branch-name>` (the -d option only deletes the branch if it has already been merged. The -D option is a shortcut for --delete --force, which deletes the branch irrespective of its merged status)
- Let's delete the branch dev using `git branch -d dev`
- Push all the changes to the remote repo using `git push`
- To delete remote dev branch, run: `git push origin :dev`

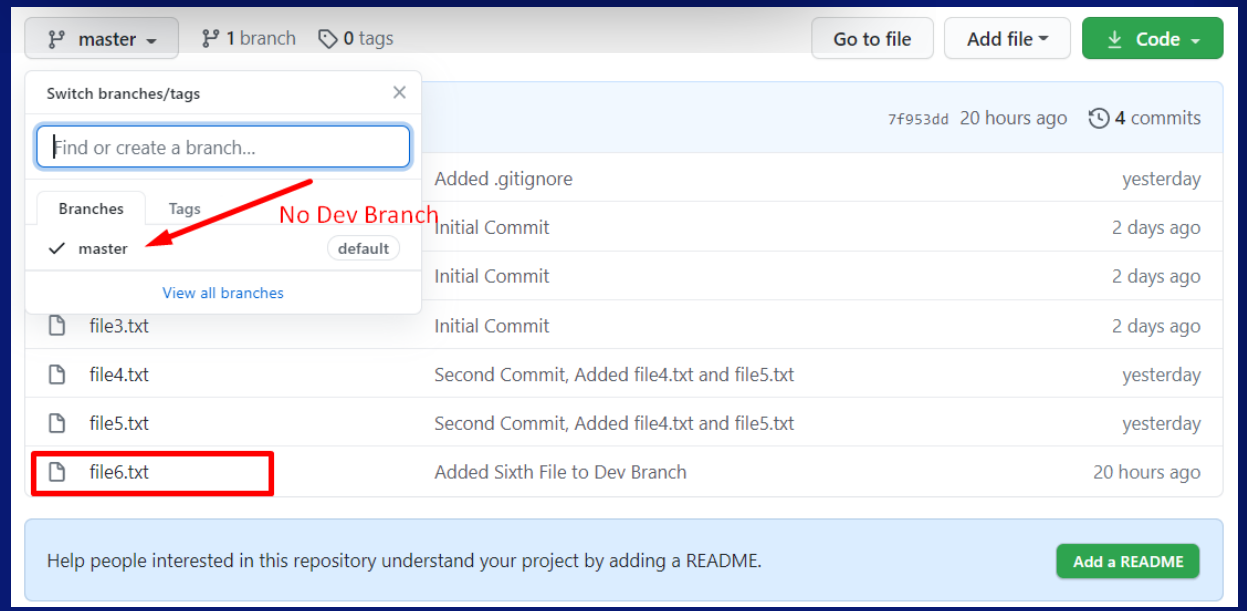


Git

Merge

- Check the remote & local branches using
  - `git branch -a`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch -a
* master
  remotes/origin/master
```




```
428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git log --oneline
7f953dd (HEAD -> dev, origin/dev) Added Sixth File to Dev Branch
755a2fc (origin/master, master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (dev)
$ git checkout master && git log --oneline
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
755a2fc (HEAD -> master, origin/master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git merge dev
Updating 755a2fc..7f953dd
Fast-forward
 file6.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
7f953dd (HEAD -> master, origin/dev, dev) Added Sixth File to Dev Branch
755a2fc (origin/master) Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt  file6.txt  TODO.txt



428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch -d dev
Deleted branch dev (was 7f953dd).

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git push
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/kunchalavikram1427/test-repo.git
   755a2fc..7f953dd  master -> master

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git push origin :dev
To https://github.com/kunchalavikram1427/test-repo.git
 - [deleted]         dev
```

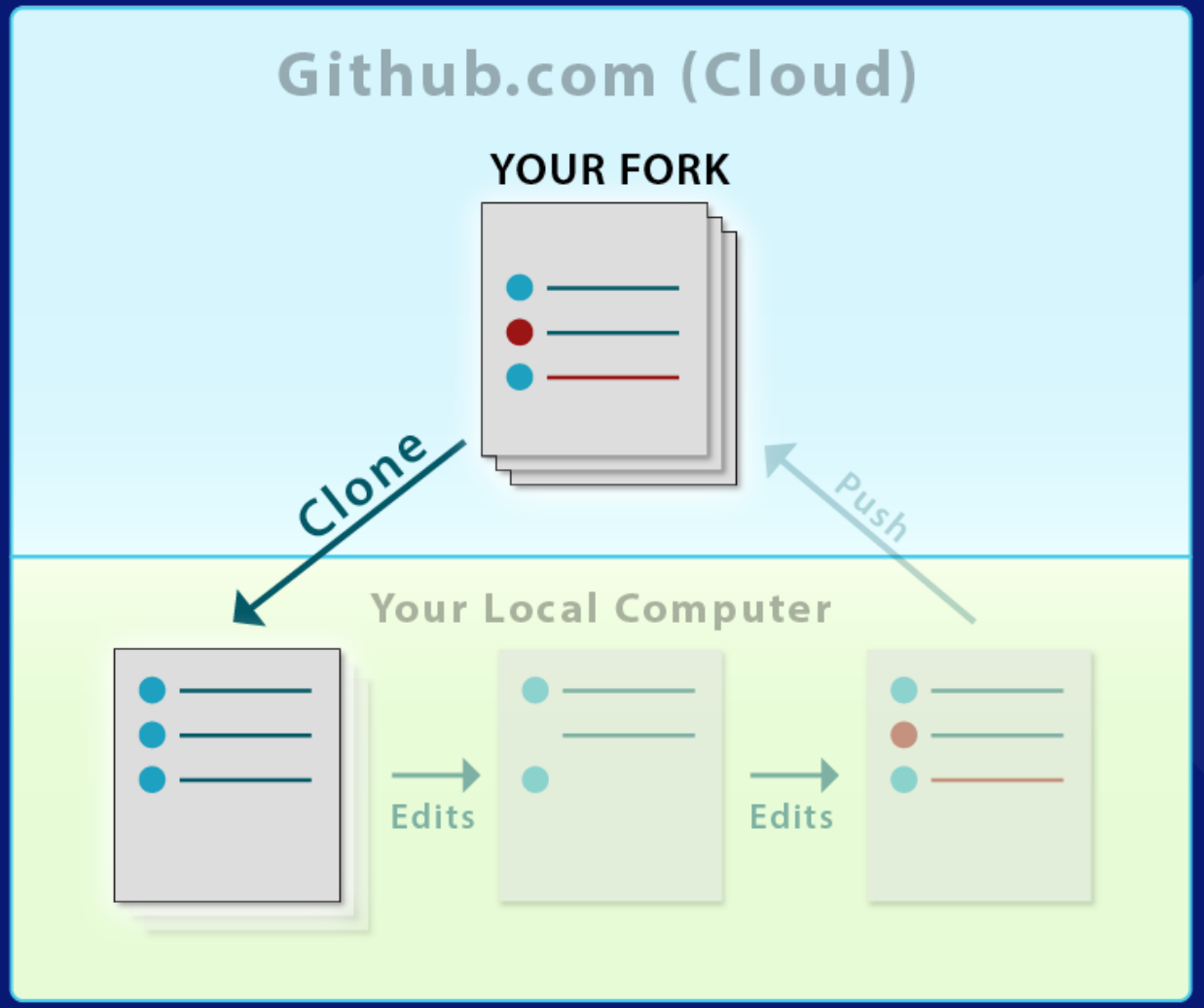
Git

Cloning an Existing Repository

- `git clone` is used to clone an existing remote repository into your computer by using the command
  - `git clone <repository-url>`
- When we clone a repository, all the files are downloaded to the local machine but the remote git repository remains unchanged
- Making changes to this cloned repo and committing them on your local repository will not affect the remote repository in any way
- These changes made on the local machine can be pushed to the remote repository anytime the user wants
- Whenever working on a team, whether if it's in an open source project or a corporate setting, it's always a good practice to create a new branch (usually based on master) and start the changes from there
- You cannot directly push your changes to the cloned repository unless you are added as the collaborator to that project by the project admin
- In any way, it is always advised to create a separate branch, work on the changes in that branch and create a Pull Request(PR) or Merge Request for merging feature branch changes to the master branch
- The master branch holds the working copy of the work product and no changes are to be pushed to that branch unless properly reviewed and approved
- For some open source projects, you cannot directly clone and push your changes(contribute). In such cases, we fork the project, make changes and submit a PR to the original project

Git

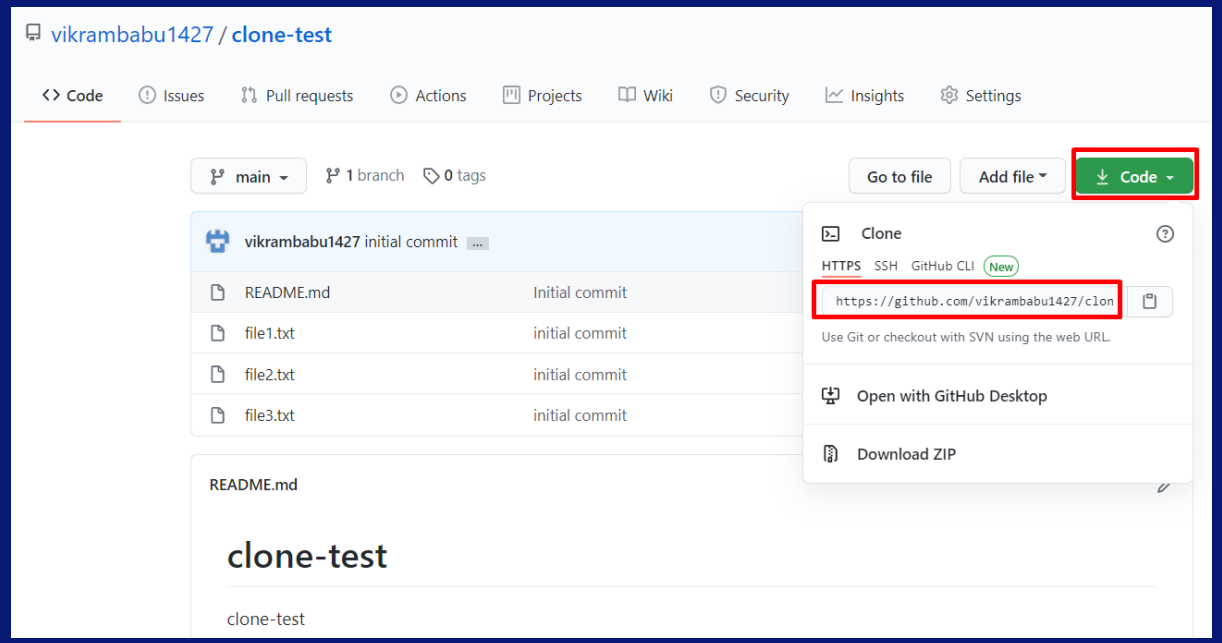
Cloning an Existing Repository



Git

Cloning an Existing Repository

- Clone the repository
  - `git clone <URL>`
- Check the existing branches
  - `git branch -a`
- Check the commit history
  - `git log --stat`




```
428991@LINL190904638 MINGW64 /d
$ git clone https://github.com/vikrambabu1427/clone-test.git
Cloning into 'clone-test'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 8 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (8/8), 1.35 KiB | 20.00 KiB/s, done.

428991@LINL190904638 MINGW64 /d
$ cd clone-test/

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  README.md

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git log --stat --summary
commit 0efb1d8d268a9f5acf74398684996ae00c9d4e3d (HEAD -> main, origin/main, origin/HEAD)
Author: vikrambabu1427 <78962677+vikrambabu1427@users.noreply.github.com>
Date:   Fri Feb 12 17:59:23 2021 +0530

    initial commit

    initial commit. Added file1,file2, file3

 file1.txt | 1 +
 file2.txt | 1 +
 file3.txt | 1 +
 3 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt

commit 3ec6e4e46a12688ece1fcac2a03a7db82ceb42
Author: vikrambabu1427 <78962677+vikrambabu1427@users.noreply.github.com>
Date:   Fri Feb 12 17:57:40 2021 +0530

    Initial commit

 README.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

Git

Cloning an Existing Repository

Lets create a new branch and make changes

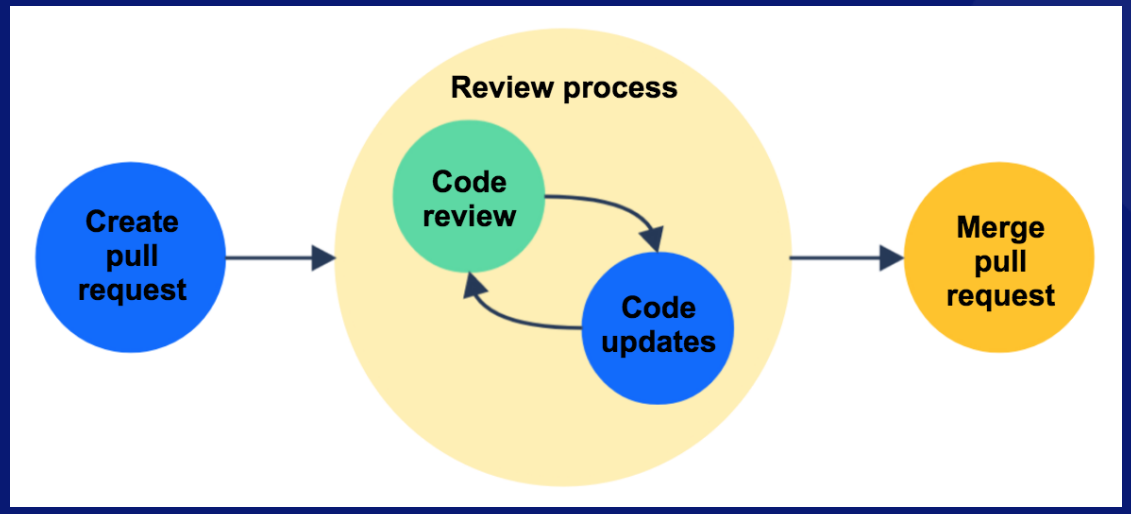
- Create new branch & switch
  - `git checkout -b feature-01`
- Edit an existing file and add it to staging
  - `git add file3.txt`
- Commit the changes to local repo
  - `git commit -m “Modified file3”`
- Push the changes to remote repo
  - `git push origin feature-01`
- Check remote & local branches
  - `git branch -a`

```
428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git checkout -b "feature-01"
Switched to a new branch 'feature-01'

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ cat file3.txt
My Third File

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ echo "Modified Third File" >> file3.txt

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git status
On branch feature-01
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file3.txt

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

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  README.md

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git add file3.txt

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git commit -m "Modified file3"
[feature-01 bedc5ec] Modified file3
 1 file changed, 1 insertion(+)

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git push origin feature-01

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 275 bytes | 137.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature-01' on GitHub by visiting:
remote:      https://github.com/vikrambabu1427/clone-test/pull/new/feature-01
remote:
To https://github.com/vikrambabu1427/clone-test.git
 * [new branch]      feature-01 -> feature-01

428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git branch -a
* feature-01
  main
  remotes/origin/HEAD -> origin/main
  remotes/origin/feature-01
  remotes/origin/main
```

Git

Create a Pull Requests

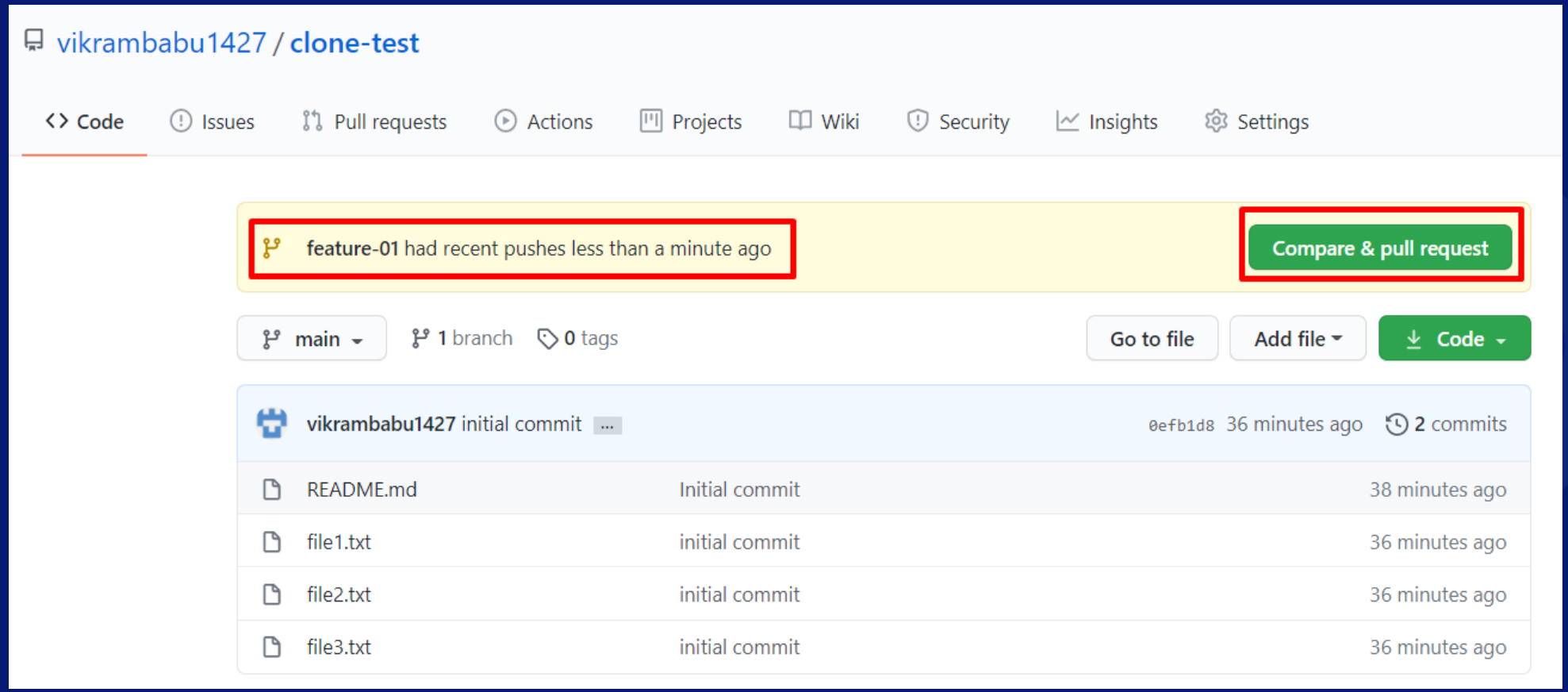
- Pull requests are a way to discuss changes before merging them into your codebase
- Let’s say you’re managing a project. A developer makes changes on a new branch and would like to merge that branch into the master. They can create a pull request to notify you to review their code. You can discuss the changes, and decide if you want to merge it or not
- Although it is possible to push and merge a branch directly into "master", creating a pull request is usually the way to go when suggesting changes in a codebase
- Opening a pull request creates an opportunity for code review and actionable feedback; that's why it became a standard procedure for collaborating in most open source projects



Git

Creating a Pull Request

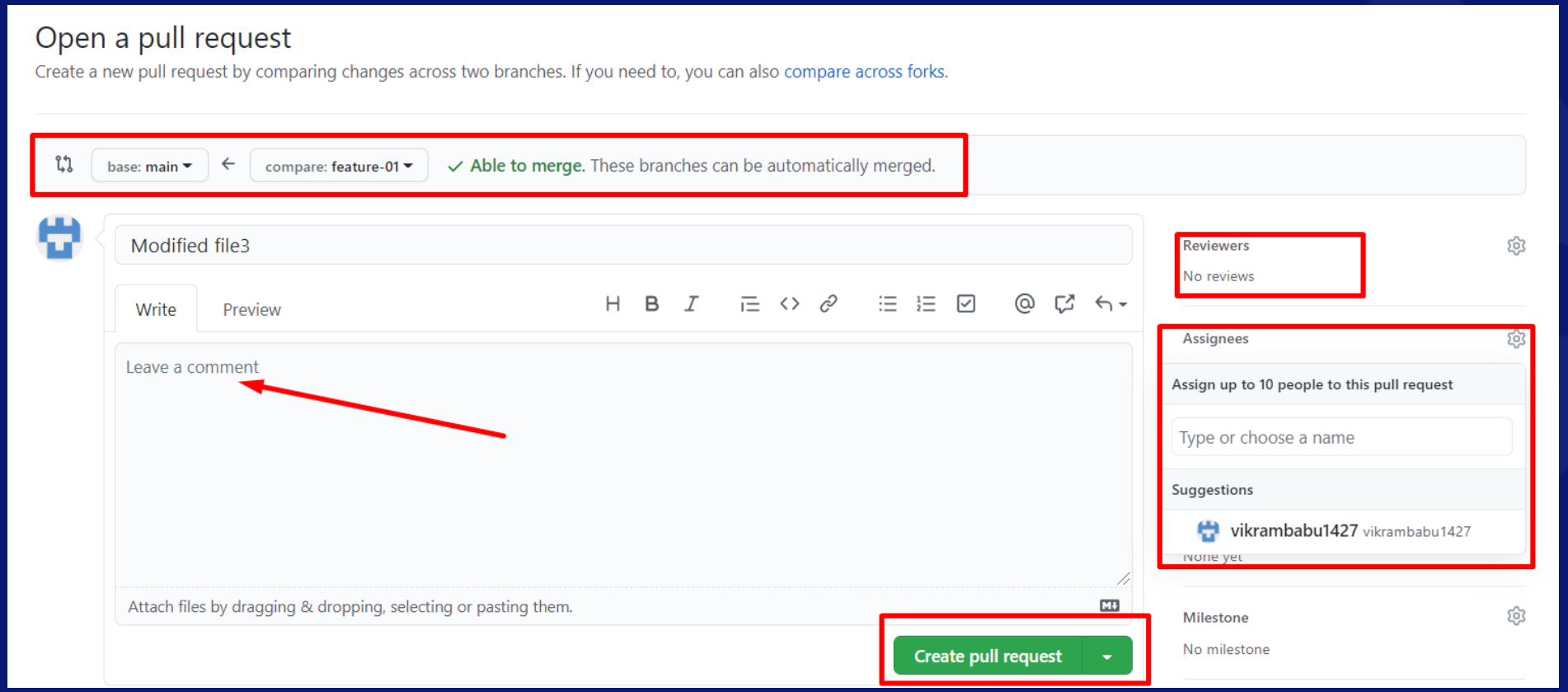
To create a PR, go to the repository and search for Create Pull Request



Git

Creating a Pull Request

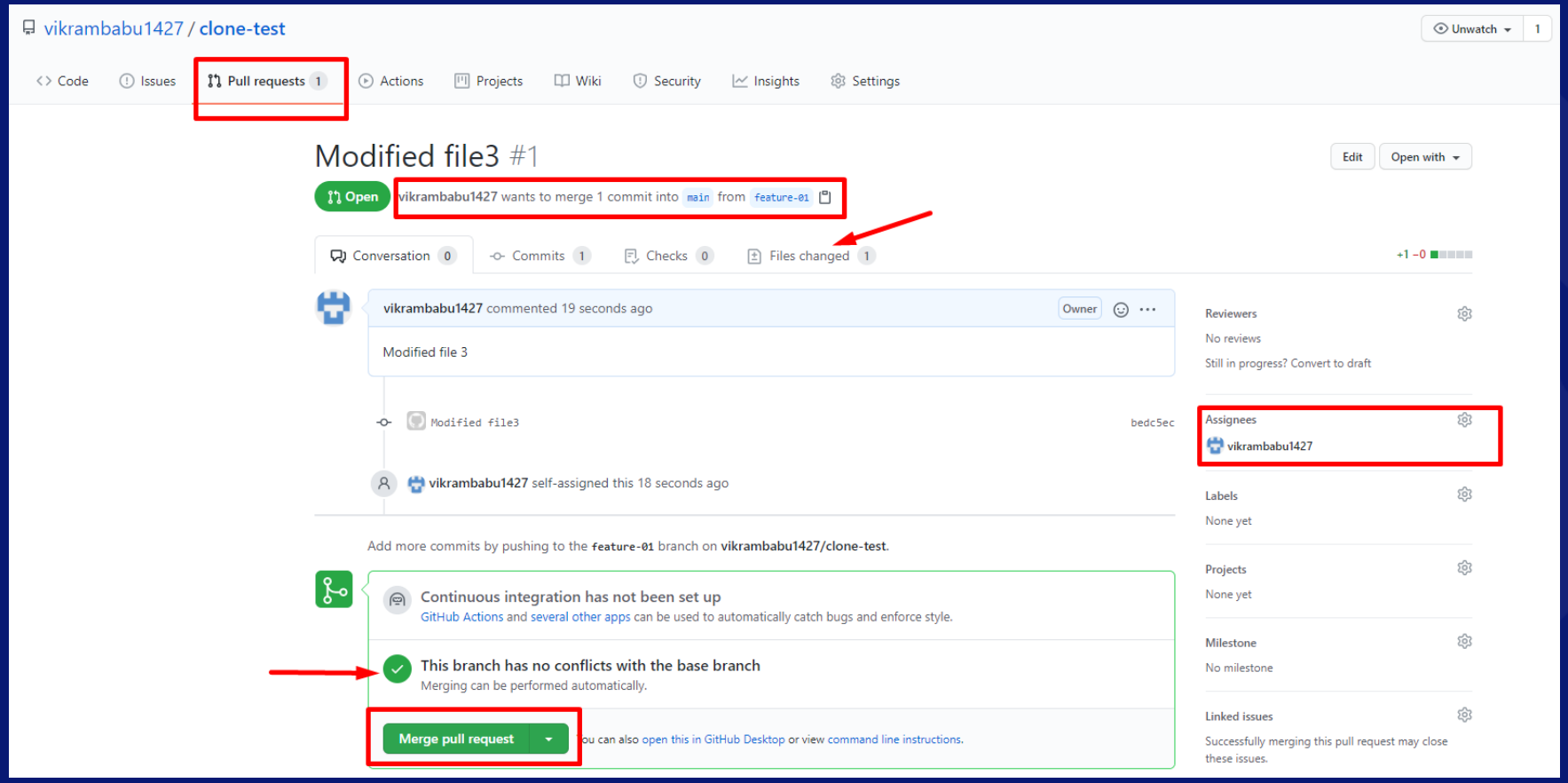
- Select the source and destination branches. In our case we want to merge feature-01 to master
- Add a Reviewer who can review the changes, Write merge comments, Select the Assignee(PR opener) and create pull request



Git

Creating a Pull Request

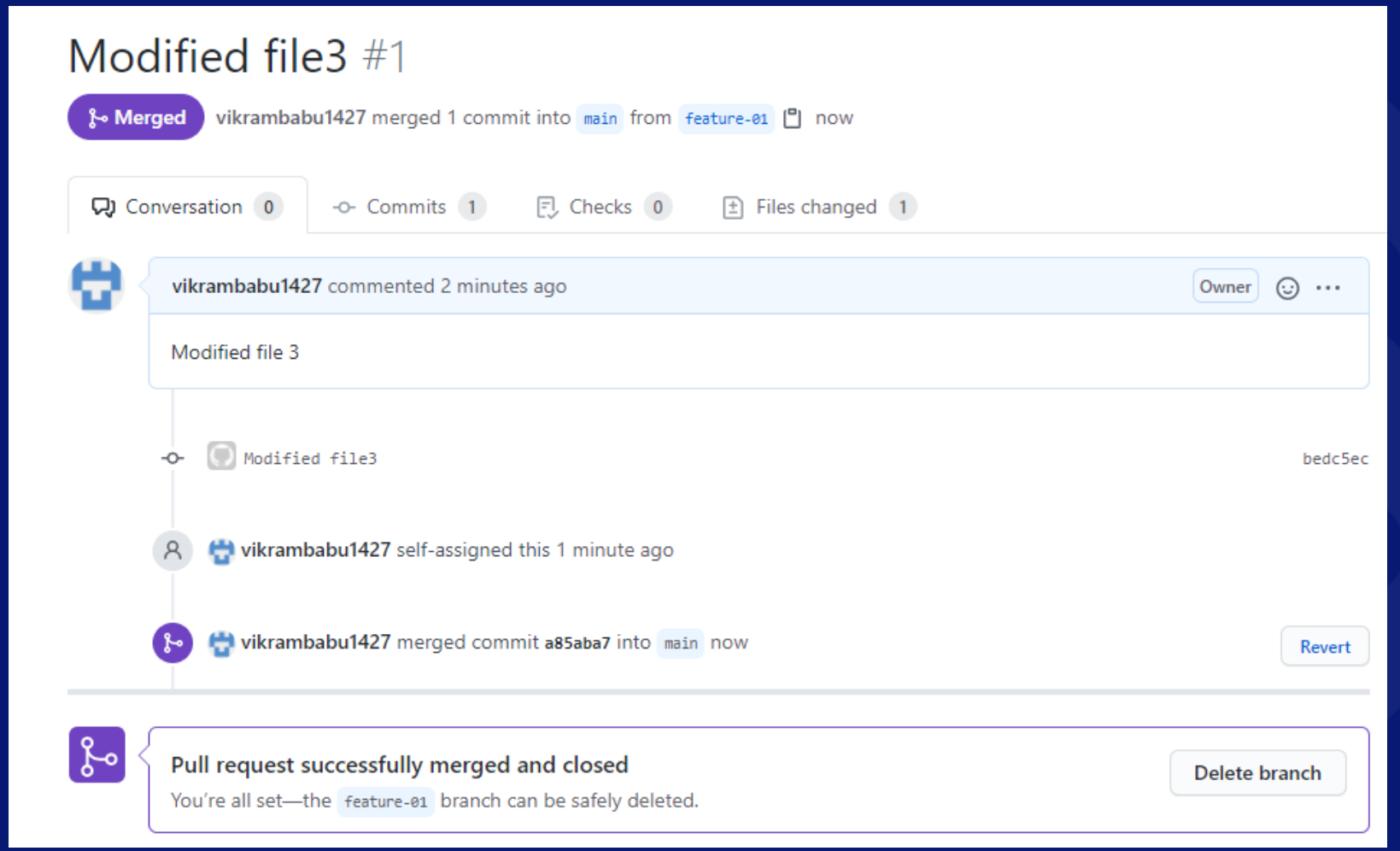
- The reviewer get notified of PR
- Verifies the files and Merges the branch if there are no conflicts. This closes the PR



Git

Creating a Pull Request

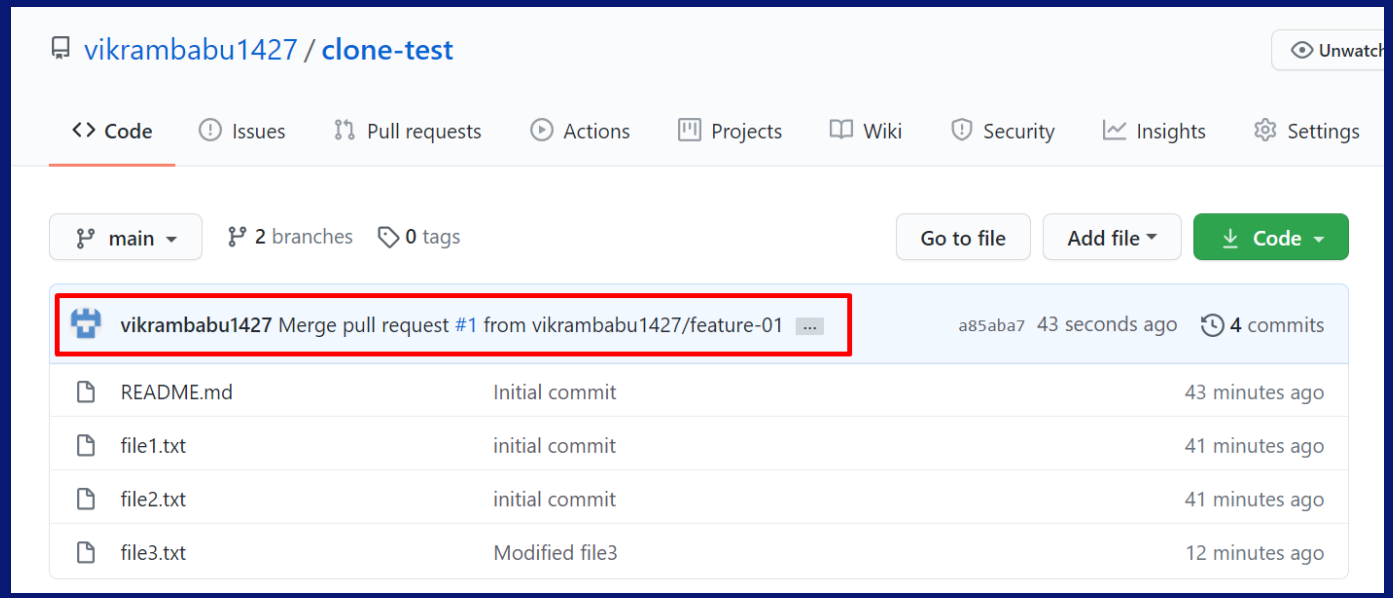
- Post successful merging

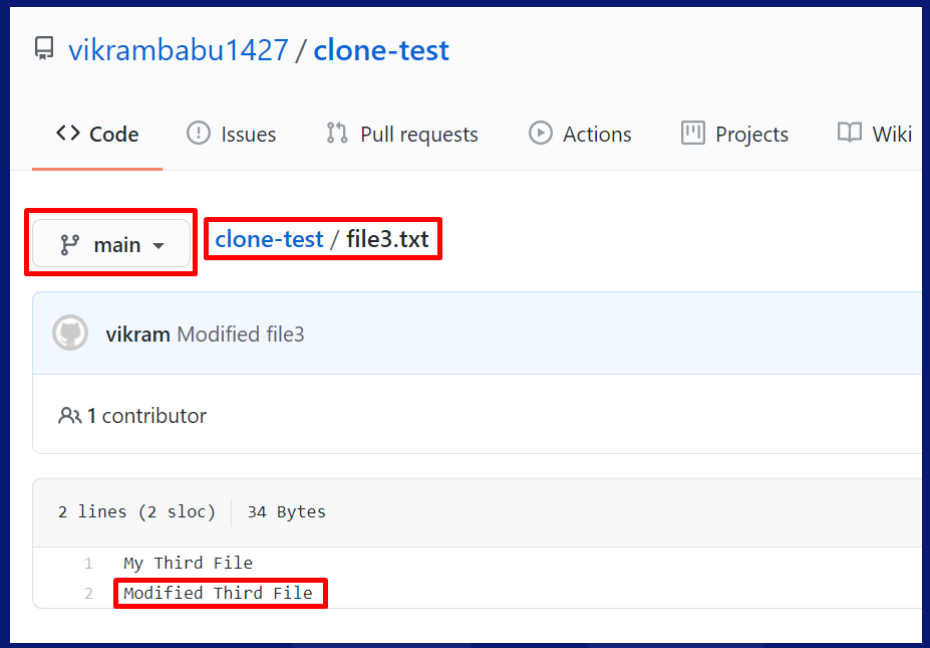


Git

Creating a Pull Request

- Post successful merging

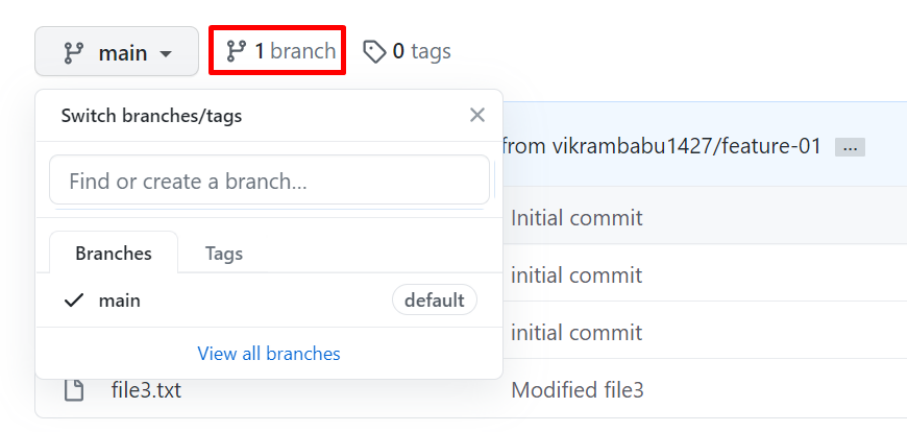




Git

Git Pull

- Pull the latest changes from the repo since there are changes in the master branch after PR
  - `git pull`
- Delete the feature branch from both local and remote as it is already merged to master
  - `git branch -d feature-01`
  - `git push origin :feature-01`

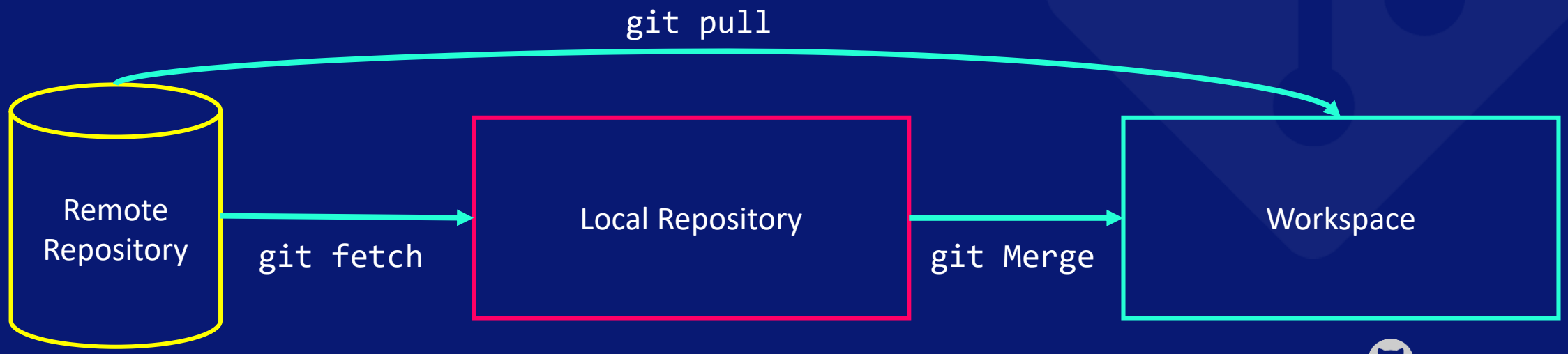



```
428991@LINL190904638 MINGW64 /d/clone-test (feature-01)
$ git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git pull
Updating 0efb1d8..a85aba7
Fast-forward
 file3.txt | 1 +
 1 file changed, 1 insertion(+)

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git branch -d feature-01
Deleted branch feature-01 (was bedc5ec).

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git push origin :feature-01
To https://github.com/vikrambabu1427/clone-test.git
 - [deleted]         feature-01
```

Git

Git Pull vs Git Fetch

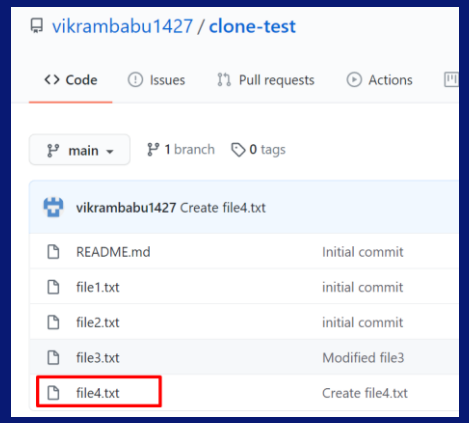
- In our previous demo, once the feature-01 branch is merged to master, master has a new modified file file3.txt
- Our local repository is unaware of this change as the merging happened directly in the remote repo
- In order to get the latest changes from the remote repo, we can use `git fetch` or `git pull`(as done in earlier demo)
- `git fetch` pulls all new commits from the desired branch and saves it in a new branch in your local repository. If you need to reflect these changes in your target branch, `git fetch` should be followed with a `git merge`
- `git pull` does both fetch and merge in 1 shot
- `Git pull = git fetch + git merge`



Git

Git Pull vs Git Fetch

- There is a new file file4.txt added to the remote repo



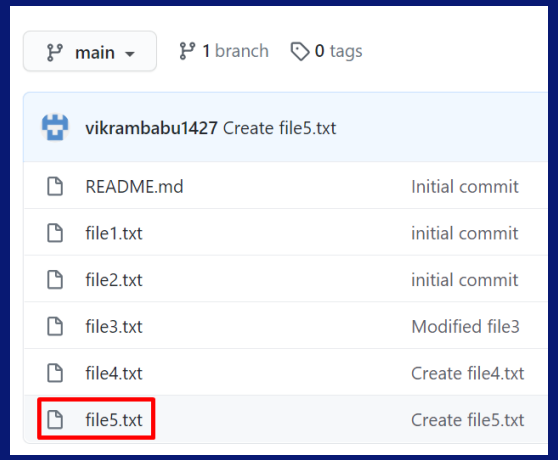
- Let's fetch all those changes into our local repository
  - `git fetch`
- List all the files in the workspace. Notice that file4.txt is still not visible
  - `ls -a`
- Lets merge the current branch with origin/main to merge the remote changes(main is master branch here)
  - `git merge origin main`
- Now List all the files in the workspace again. Notice file4.txt added to the workspace
  - `ls -a`

```
428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
From https://github.com/vikrambabu1427/clone-test
   a85aba7..9261946  main       -> origin/main

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  README.md

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git merge origin master
merge: master - not something we can merge

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git merge origin main
Updating a85aba7..9261946
Fast-forward
 file4.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file4.txt

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  file4.txt  README.md
```

Git

Git Pull vs Git Fetch

- There is a new file file5.txt added to the remote repo



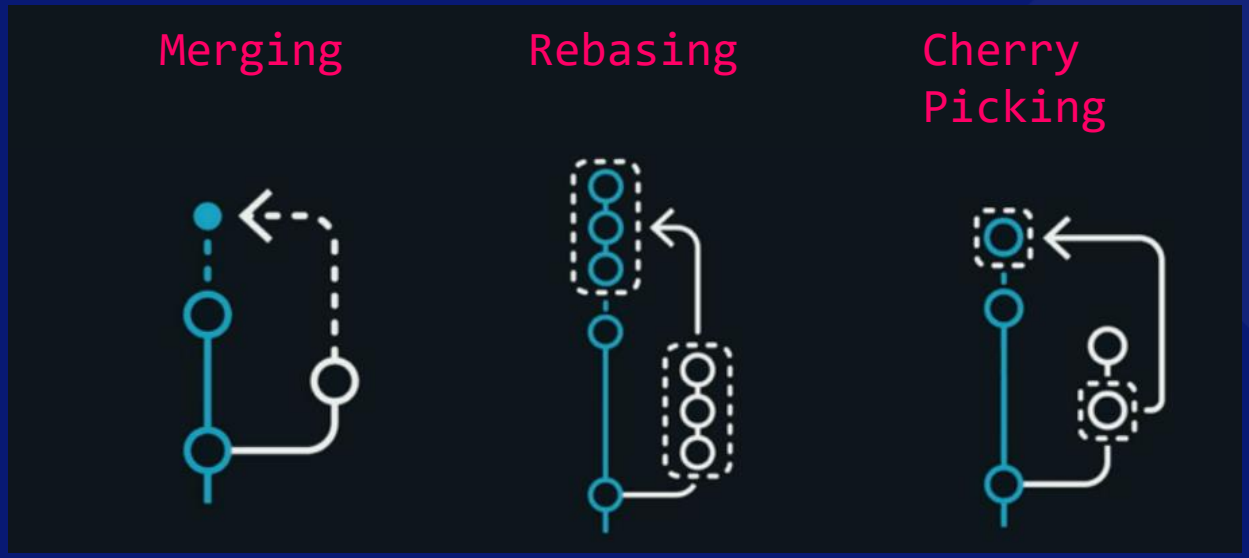
- List all the files in the workspace. Notice that file5.txt is still not visible
  - `ls -a`
- Lets pull all the changes
  - `git pull origin main`
- Now List all the files in the workspace again. Notice file5.txt added to the workspace
  - `ls -a`

```
428991@LINL190904638 MINGW64 /d/clone-test (main)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  file4.txt  README.md

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git pull origin main
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 669 bytes | 41.00 KiB/s, done.
From https://github.com/vikrambabu1427/clone-test
 * branch            main       -> FETCH_HEAD
   9261946..9a17738  main       -> origin/main
Updating 9261946..9a17738
Fast-forward
 file5.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file5.txt

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ ls -a
./  ../  .git/  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt
README.md

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ cat file5.txt
Added Fifth File
```

Git

Merge Conflicts

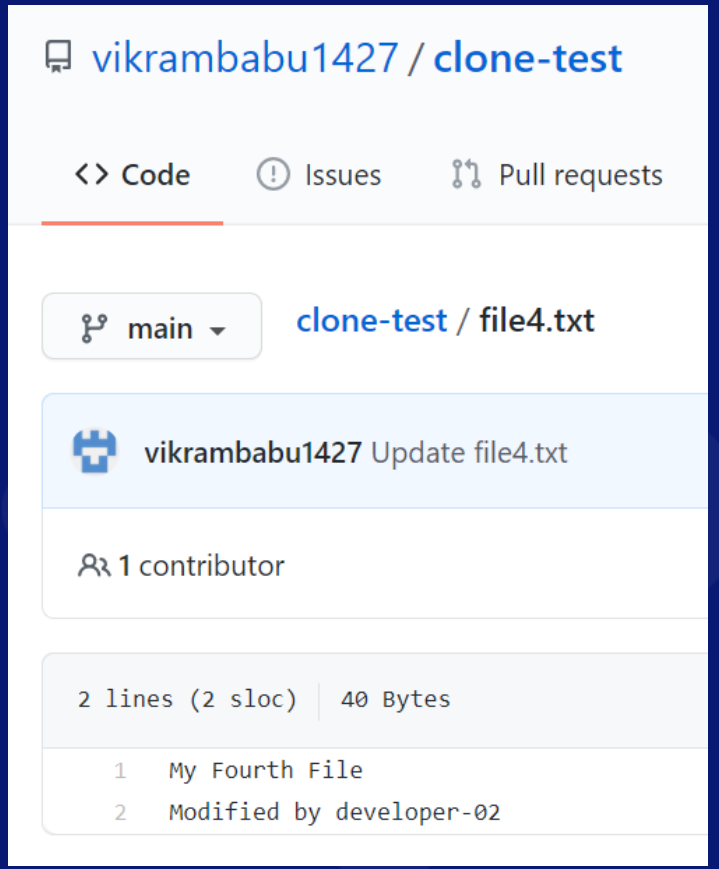
- A merge conflict is an event that occurs when Git is unable to automatically resolve differences in code between two commits
- Since working with remote repositories allows multiple developers to work in parallel, each on their copy of the repository, merge conflicts may occur when two developers change the same file and try to upload the changes to the central repo.
- The way to solve the problem is by pulling the updated version from the central repository, updating the file with your changes, and pushing it back to remote repository
- Conflicts occur during



Git

Merge Conflicts

- To understand merge conflicts, let's modify the file file4.txt in the remote repo, replicating a real time scenario
- Since the file is changed in the remote repo, our local repo is unaware of these changes as we didn't do `git pull` to update it
- Let us now modify the same file file4.txt in our local and try to push it to remote repo
- Since the same file is already modified in the remote repo(pushed by developer-02), there will be merge conflicts and push to remote fails
- Git will show us the list of files that are conflicting with the remote repo
- We need to modify these files by deleting or retaining the content after discussing with the developer who committed the change before
- After the modifications, we can commit those changes and push to remote repository



Git

Merge Conflicts

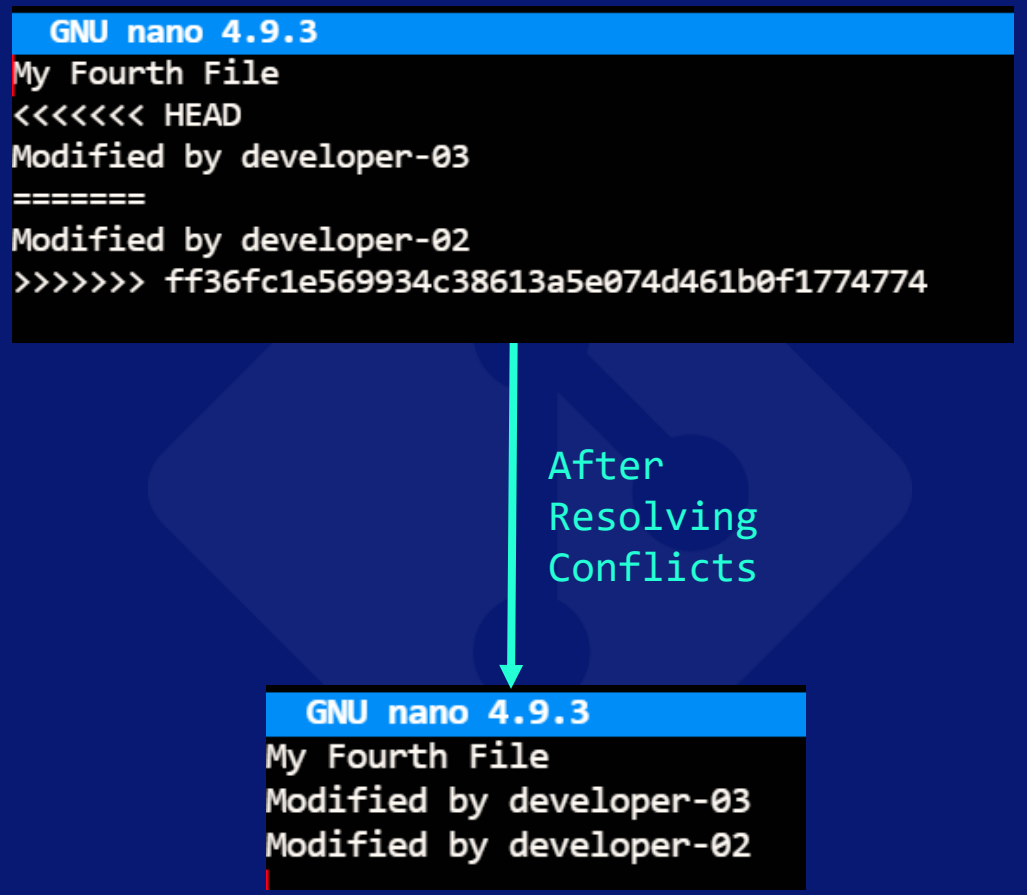
- There is a new file file4.txt added to the remote repo by other developer
- To the same file in our local, add some content and commit it to the local repo
  - `git commit -a -m "Modified file4 by dev 03"`
- Push the changes to the remote origin
  - `git push`
- Observe that the push failed due to merge conflicts. Developer-02 has already modified the same file and pushed it to remote before developer-03 pushed the same file
- Git will show us the list of files with merge conflicts
  - `git status`
- We need to modify these files to include common changes needed by both the developers

```
428991@LINL190904638 MINGW64 /d/clone-test (main)
$ echo "Modified by developer-03" >> file4.txt

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ cat file4.txt
My Fourth File
Modified by developer-03

428991@LINL190904638 MINGW64 /d/clone-test (main|MERGING)
$ git commit -a -m "Modified file4 by dev 03"
[main 75ce4c6] Modified file4 by dev 03

428991@LINL190904638 MINGW64 /d/clone-test (main|MERGING)
$ git push
To https://github.com/vikrambabu1427/clone-test.git
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to
'https://github.com/vikrambabu1427/clone-test.git'
hint: Updates were rejected because the tip of your current branch is
behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.

428991@LINL190904638 MINGW64 /d/clone-test (main|MERGING)
$ git status
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file4.txt
```

Git

Resolve Merge Conflicts

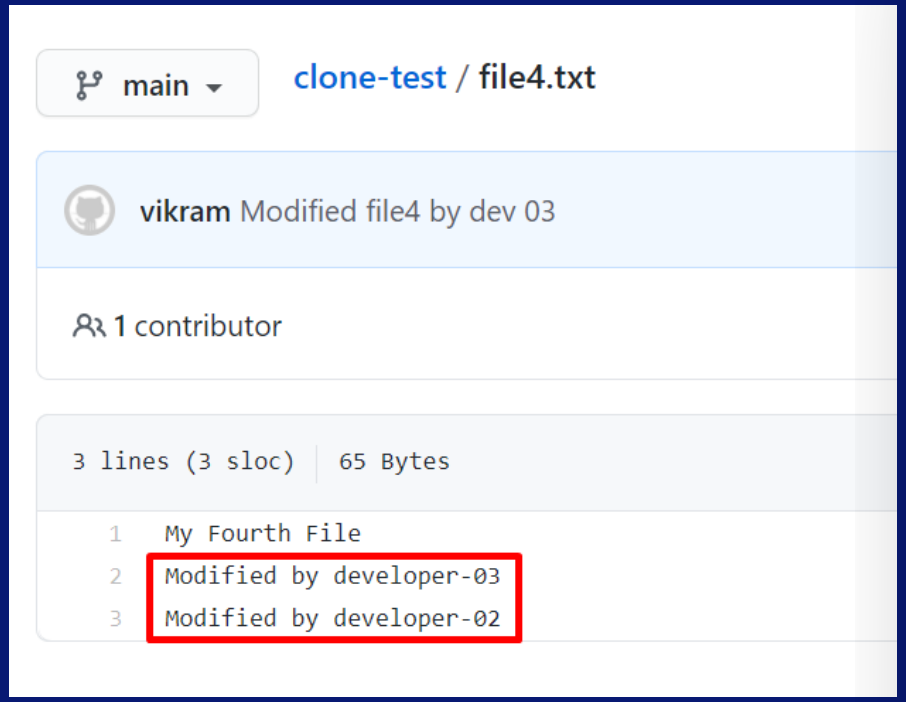
- Open the file which has merge conflicts in any of the text editors
- Git added some syntax including seven "less than" characters, `<<<<<<<` and seven "greater than" characters, `>>>>>>>`, separated by seven equal signs, `=======`. These can be searched using your editor to quickly find where edits need to be made.
- There are two sections within this block:
  1. The "less than" characters denote the current branch's edits (in this case, "HEAD," which is another word for your current branch), and the equal signs denote the end of the first section.
  2. The second section is where the edits are from the attempted merge; it starts with the equal signs and ends with the "greater than" signs
- As a developer, you decide what stays and what goes
- We need to remove these less/greater than characters & equal signs and keeps the changes required. In our case, we decide to include both the changes
- Now commit the modified files and push to the remote repo
  - `git commit -a -m "Modified file4 by dev 03"`
  - `git push`



Git

Resolve Merge Conflicts

- Check the changes in the remote repository as well

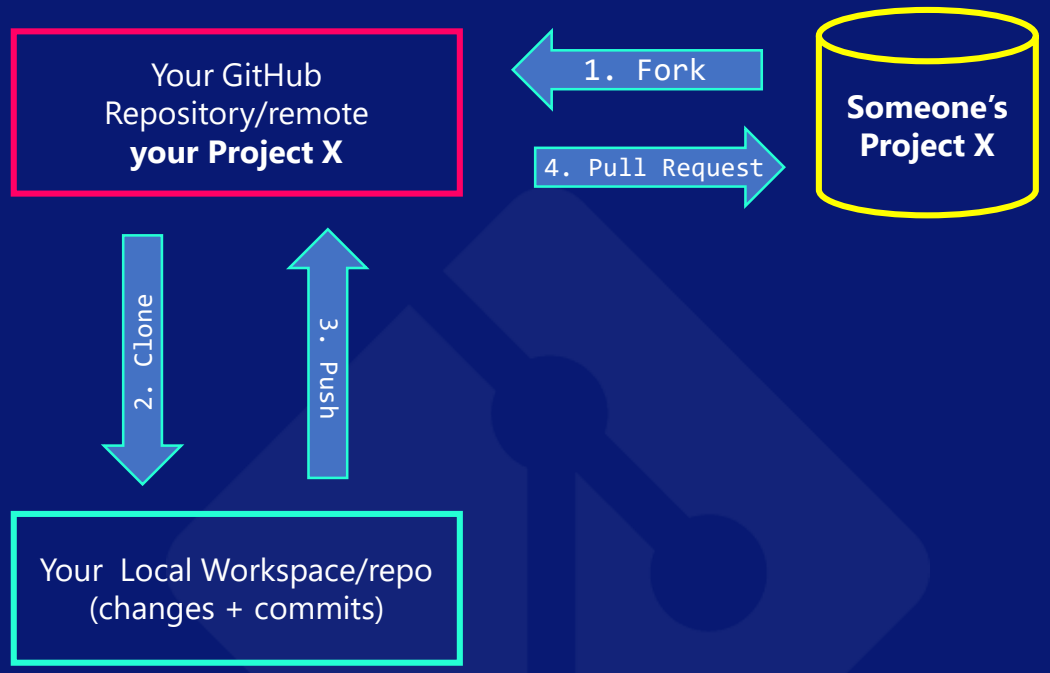



```
428991@LINL190904638 MINGW64 /d/clone-test (main|MERGING)
$ cat file4.txt
My Fourth File
Modified by developer-03
Modified by developer-02

428991@LINL190904638 MINGW64 /d/clone-test (main|MERGING)
$ git commit -a -m "Modified file4 by dev 03"
[main 75ce4c6] Modified file4 by dev 03

428991@LINL190904638 MINGW64 /d/clone-test (main)
$ git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 595 bytes | 297.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/vikrambabu1427/clone-test.git
   ff36fc1..75ce4c6  main -> main
```

Git

Fork vs Clone

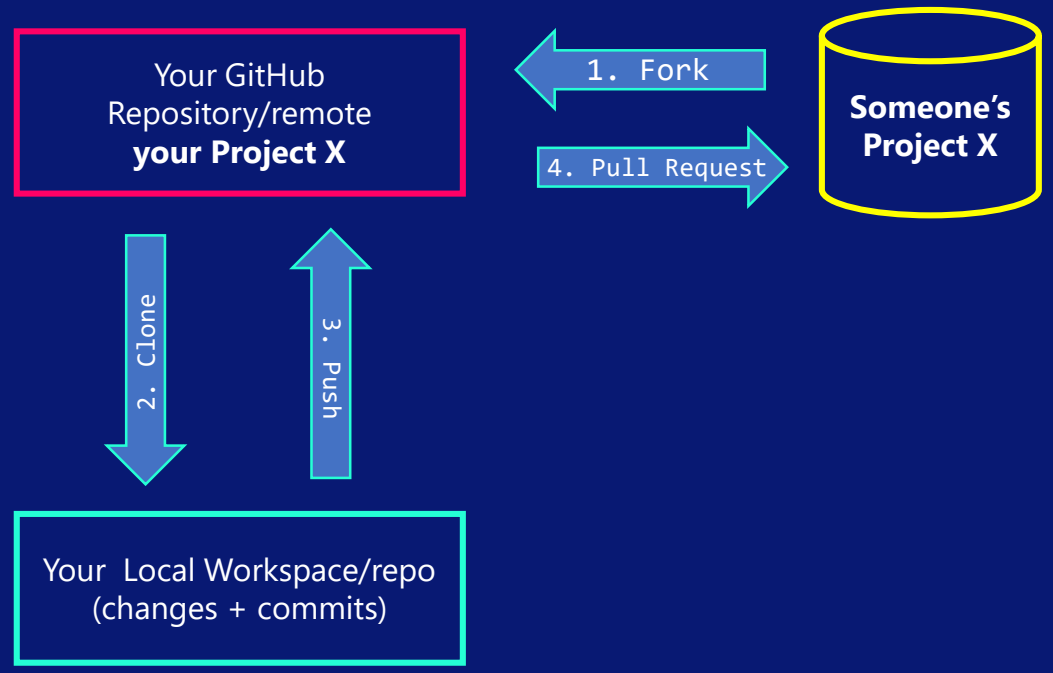
- Put simply, a `fork` is a copy of a repository that sits in your account rather than the account from which you forked the data from. Once you have forked a repo, you own your forked copy. This means that you can edit the contents of your forked repository without impacting the parent repo
- The fork will also "know" from which repository it has been forked
- Your fork acts as a bridge between the original repository and your personal copy where you can contribute back to the original project using Pull Requests
- The workflow usually includes Forking a repo owned by someone into your GitHub account, Clone the fork of your repo to your local machine, make edits, commit and push those edits back to your fork on GitHub and create a PR to original source
- Whatever the mechanism behind the fork really is, all VCS tools like GitHub, Bitbucket and GitLab all offer forking functionality
- Forking a project is as easy as clicking the `Fork` button in the header of a repository. Once the process is complete, you'll be taken right to your the forked copy of the project so you can start collaborating



Git

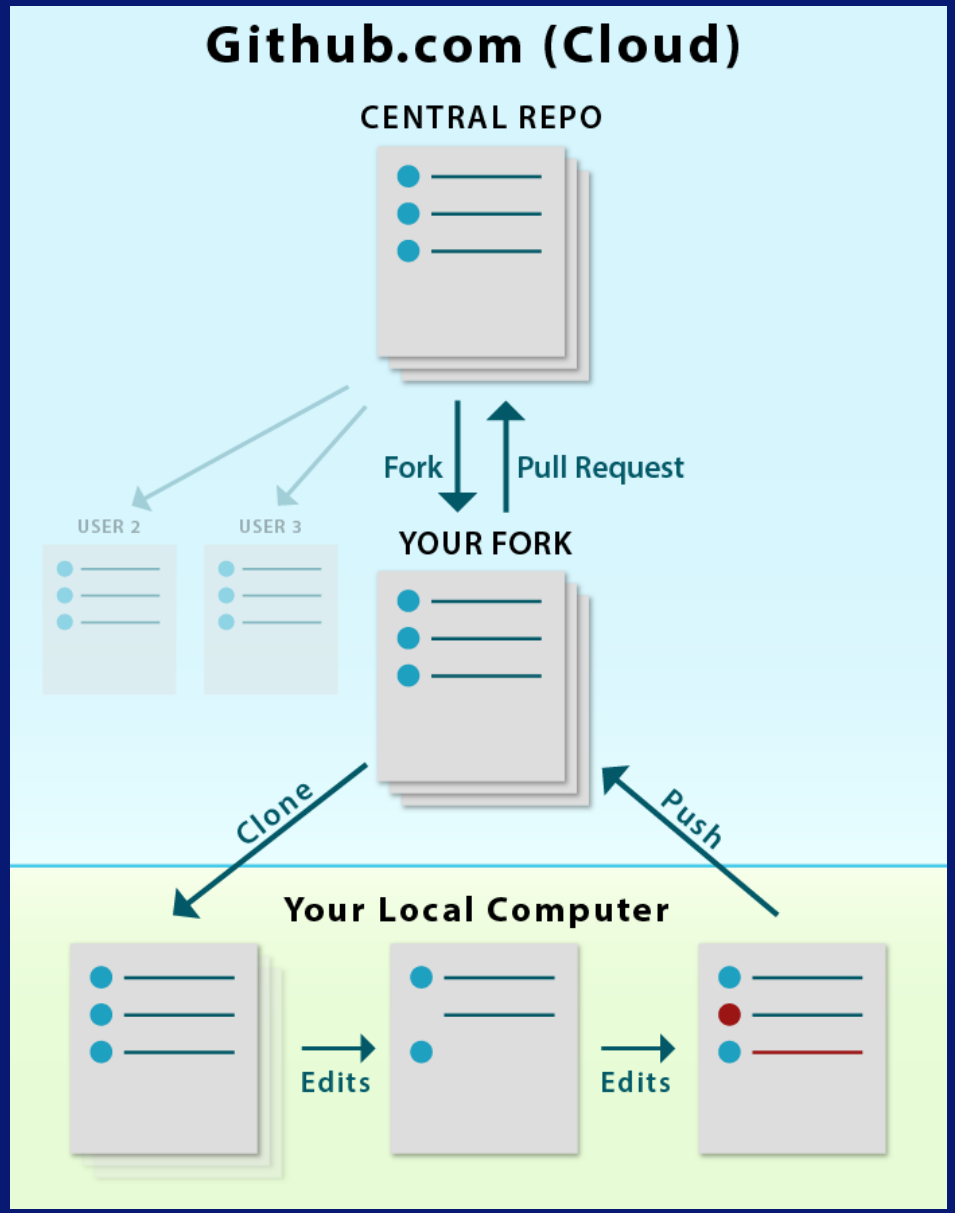
Fork vs Clone

- When you create a new repository on GitHub, it exists as a remote location where your project is stored
- You can `clone` your repository to create a local copy on your computer so that you can sync between both the local and remote locations of the project
- Unlike forking, if the project is owned by someone else you won't be able to contribute back to it unless you are specifically added as a collaborator
- Cloning is ideal for instances when you need a way to quickly get your own copy of a repository where you may not be contributing to the original project



Git

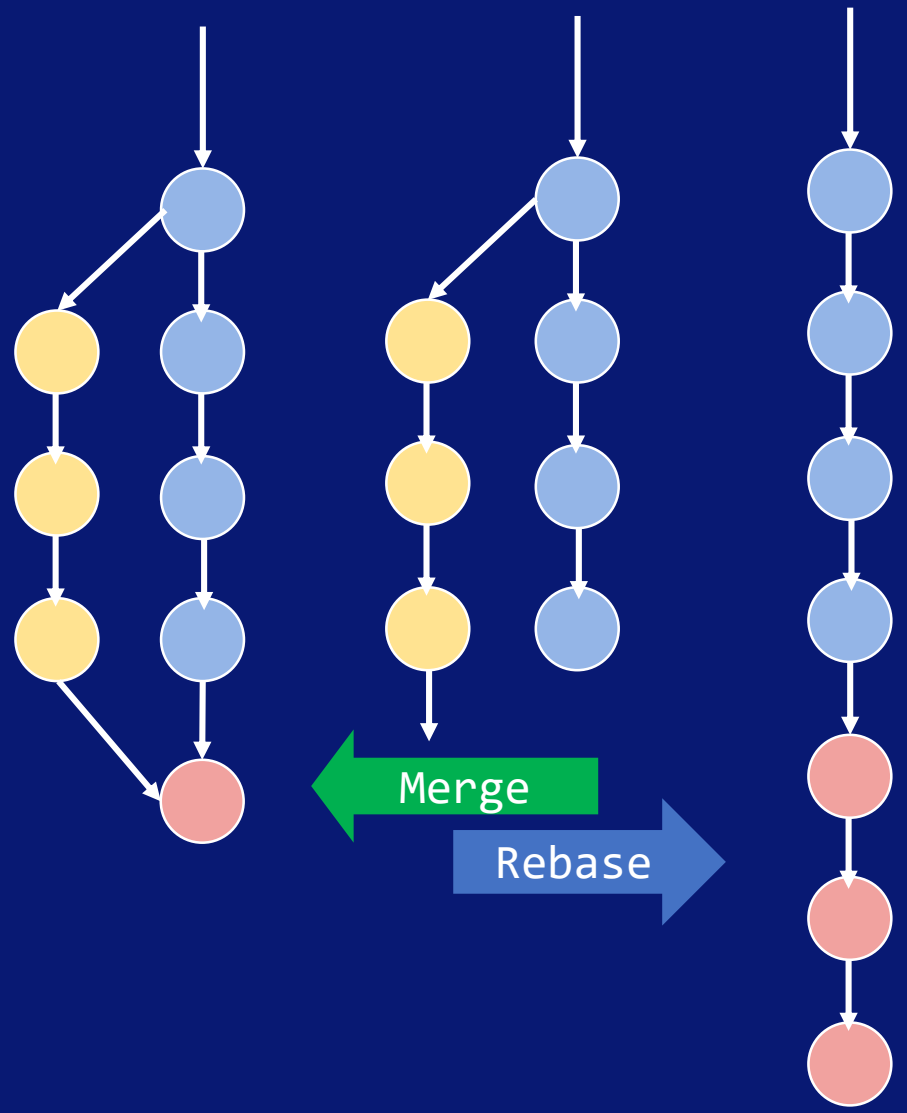
Fork vs Clone



Git

Rebase

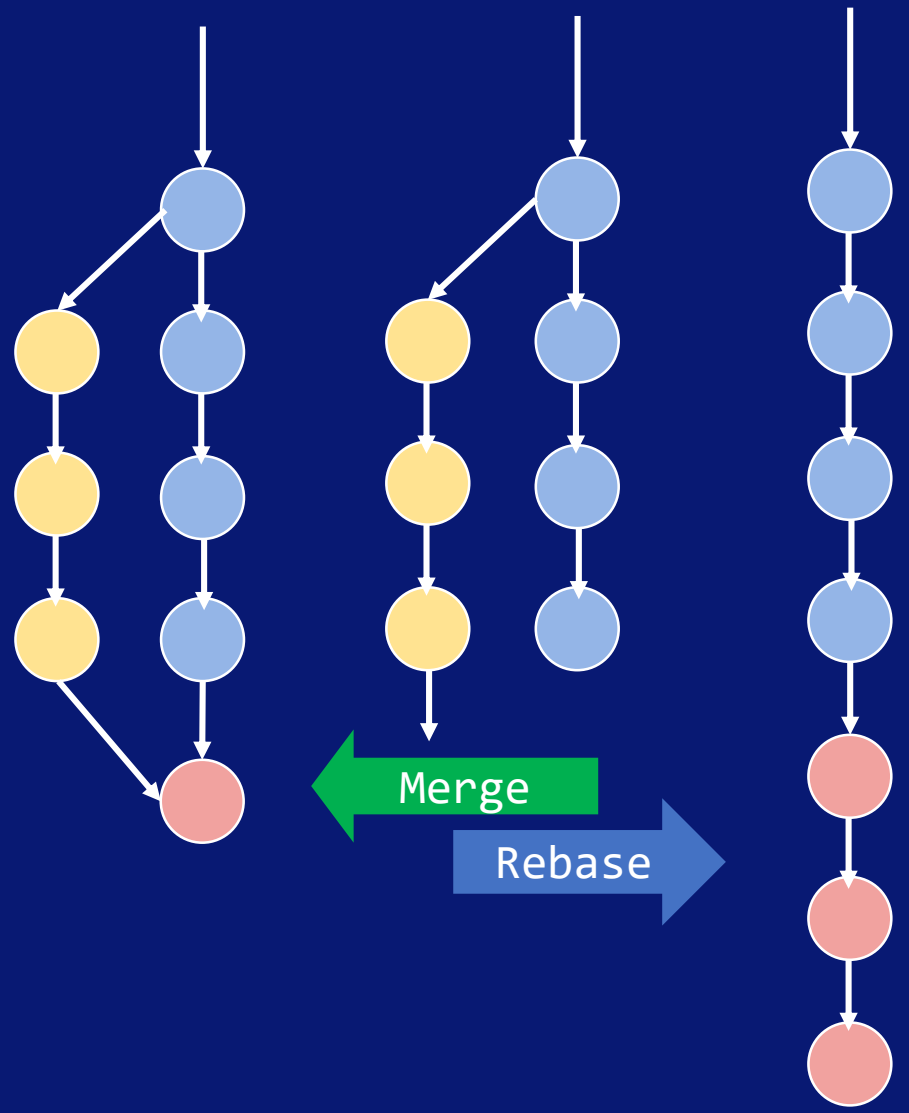
- There are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`
- A rebase is an alternative to a merge for combining multiple branches
- In a `merge`, if both branches have changes, then a new merge commit is created, leaving a non-linear history
- In `rebasing`, Git will take the commits from one branch and replay them, one at a time, on the top of the other branch, leaving a linear history
- In essence, rebase is an automated way of performing several `cherry-picks` in a row
- Git Rebase also allows us to rewrite Git history. It is a common practice to use git rebase to squash commits before creating or merging a pull request. Ex: Nobody needs to see that you fixed 10 typos in 5 separate commits, and keeping that history is of no use. We can combine these individual commits into 1 single commit



Git

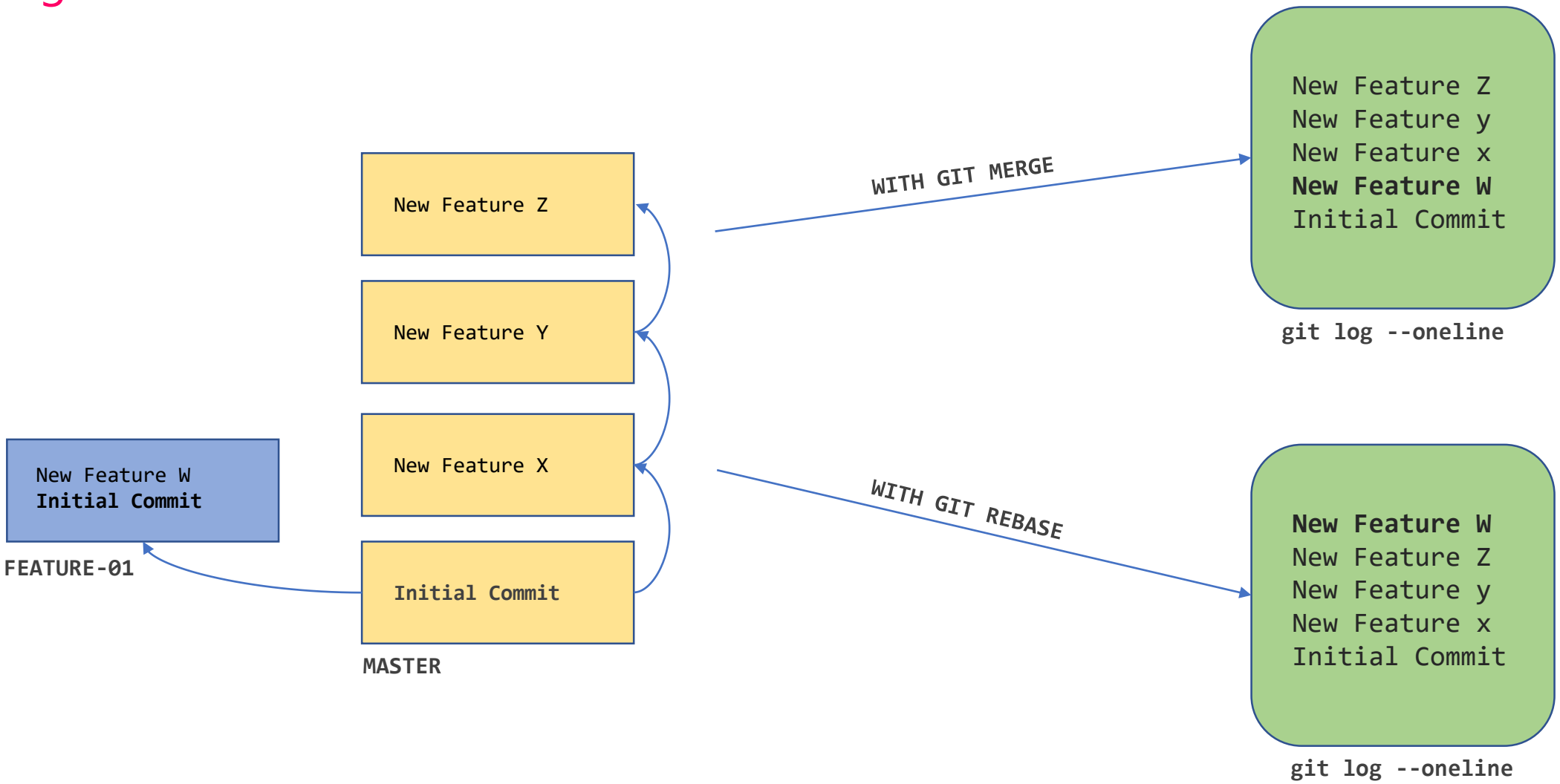
Merge vs Rebase

- `git merge` leaves a non linear commit, meaning git will have commits ordered by the time of creation and not by time of insertion, whereas `git rebase` orders by the (real) time of the insertion of the commit
- When Git Rebasing a branch on top of master (for example), Git will put aside your new commits of that branch (commits that are not part of master just yet), then it will sync-up the branch compared to master and only then bring back your new commits
- So the new commits will be always on top



Git

Merge vs Rebase

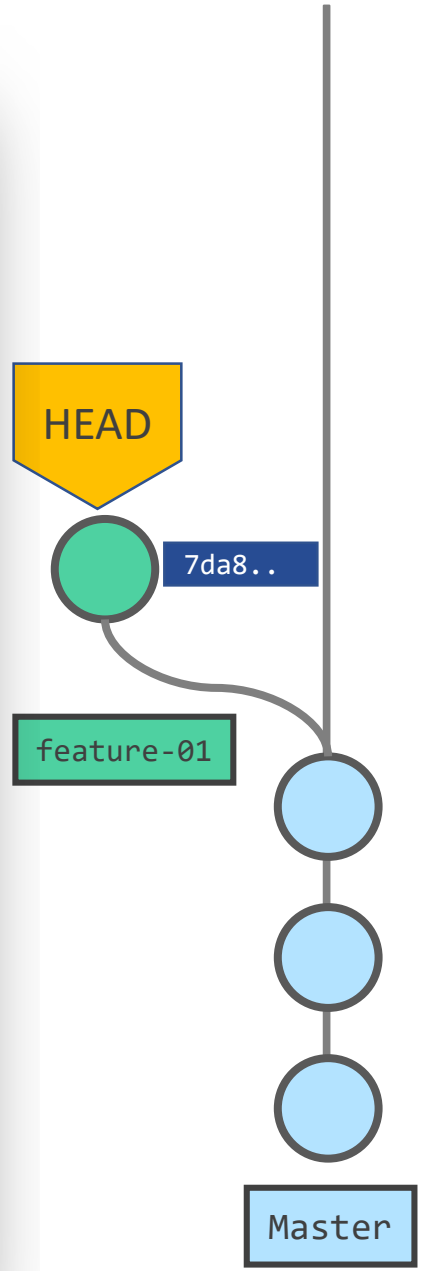


Git

Rebase

- Lets create a new branch feature-01 from master
  - `git checkout -b feature-01`
- Modify file6.txt as shown in the image and commit it to the local repo
  - `git commit -am "Modified file6 in feature-01"`
- Check the commit log
  - `git log --one-line`
- Check HEAD (HEAD points out the last commit in the current checkout branch)
  - `git show head`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git branch -a
* master
  remotes/origin/master

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt  file3.txt  file4.txt
file5.txt  file6.txt  TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git checkout -b feature-01
Switched to a new branch 'feature-01'

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ cat file6.txt
My Sixth File

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ echo "Modified file6 in feature-01" >> file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git status
On branch feature-01
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file6.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git commit -am "Modified file6 in feature-01"
warning: LF will be replaced by CRLF in file6.txt.
The file will have its original line endings in your working directory
[feature-01 7da8ab5] Modified file6 in feature-01
 1 file changed, 1 insertion(+)

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --online
fatal: unrecognized argument: --online

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
7da8ab5 (HEAD -> feature-01) Modified file6 in feature-01
7f953dd (origin/master, master) Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```



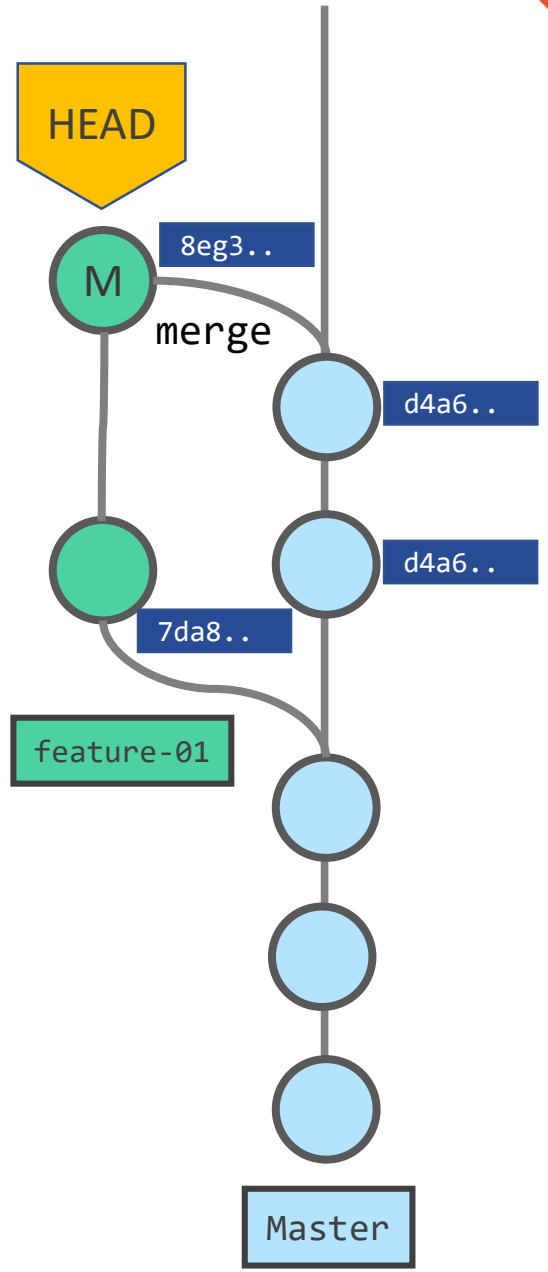
Git

Rebase

- Let's say before feature-01 branch is pushed to the remote repo, other developers pushed the latest code to the master branch. Check the GitHub and see the commit history
- Now there are two ways you can integrate the latest changes from Master to the feature-01 branch
  1. Merge master to Feature-01
     - `git checkout feature-01`
     - `git merge master`
  2. Rebase the feature-01 branch
     - `git checkout feature-01`
     - `git rebase master`

Using Merge

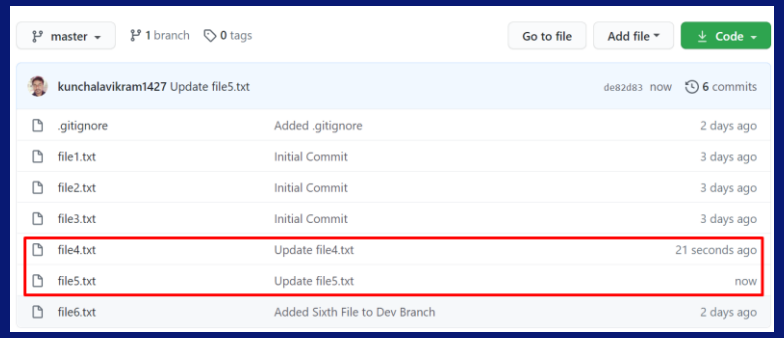
- With merging as shown in the image, we merge the latest changes from master to feature-01 branch every time there is a change
- On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch's history quite a bit



Git

Rebase

- There are some changes in the master branch in the remote repository. Let's pull all the latest changes to our local repository
  - `git checkout master`
  - `git pull origin master`



- Check the commit history in the master
  - `git log --oneline`
- Similarly Check the commit history in our local feature-01 branch
  - `git checkout feature-01`
  - `git log –oneline`
- You can see from the log that master is 2 commits ahead of feature-01

```
428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git pull origin master
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.33 KiB | 56.00 KiB/s, done.
From https://github.com/kunchalavikram1427/test-repo
 * branch            master     -> FETCH_HEAD
   7f953dd..de82d83  master     -> origin/master
Updating 7f953dd..de82d83
Fast-forward
 file4.txt | 1 +
 file5.txt | 1 +
 2 files changed, 2 insertions(+)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
de82d83 (HEAD -> master, origin/master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git checkout feature-01
Switched to branch 'feature-01'

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
7da8ab5 (HEAD -> feature-01) Modified file6 in feature-01
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```

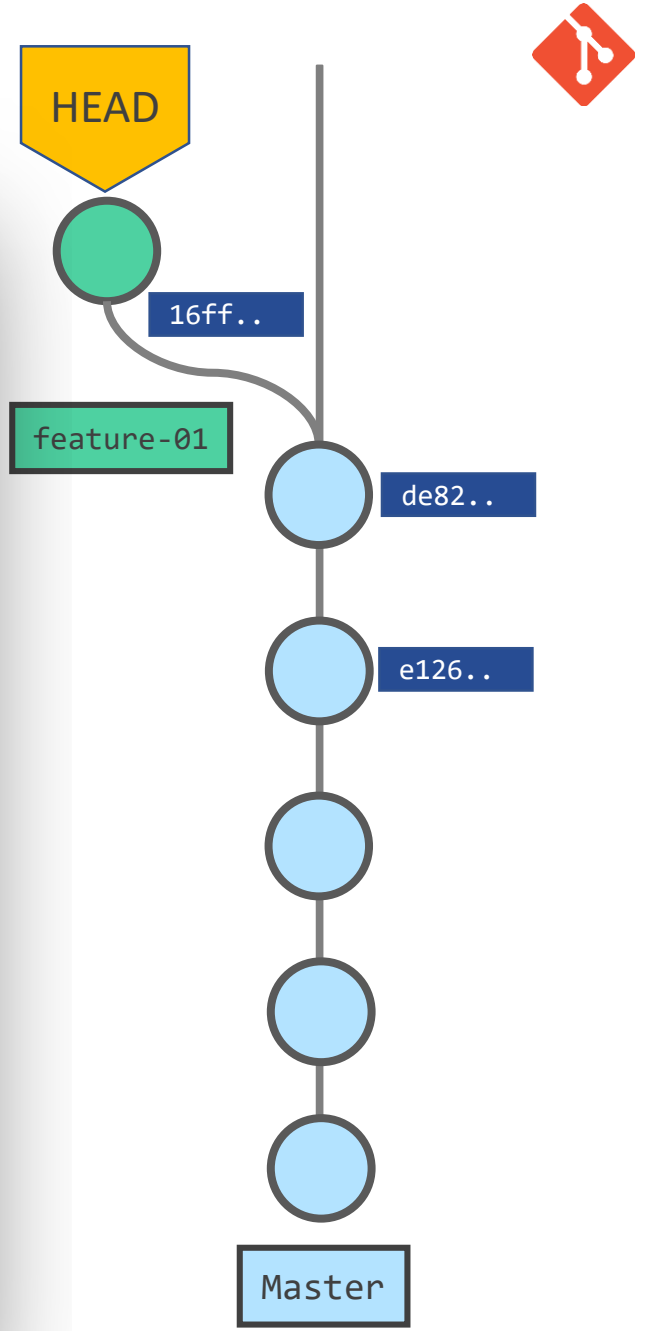
Git

Rebase

Using Merge

- Let's include all the latest changes from the master. But this time instead of merging master to feature-01, lets rebase the feature-01 branch
- Rebase the feature-01 branch
  - `git checkout feature-01`
  - `git rebase master`
- Let's check the commit history carefully. In Merging, git will merge the changes and create a new commit. But in case of rebasing, Git will rebase our current branch.

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
de82d83 (HEAD -> master, origin/master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git checkout feature-01
Switched to branch 'feature-01'

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
7da8ab5 (HEAD -> feature-01) Modified file6 in feature-01
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git rebase master
Successfully rebased and updated refs/heads/feature-01.

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
16ff49d (HEAD -> feature-01) Modified file6 in feature-01
de82d83 (origin/master, master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```



Interactive Rebasing

- Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch
- It can alter branch's commit history to clean up its messy history before merging a feature branch into master
- Sometimes a feature implementation can have multiple commits; including changes, typo corrections, temporary commits etc
- But In order to keep your master commits log nice and clean, it's important to have one commit per feature (or bug fix). You can have 5, 10 or 100 commits in your feature branch, but once it's ready to be merged into master, it's highly recommended to squash those commits into a nice clean single commit
- When running an interactive rebase with `rebase -i`, there are mainly three actions we may want to perform in a commit:
  1. `pick: pick a commit.`
  2. `squash: squash this commit into the previous one.`
  3. `drop: drop this commit altogether.`
- Interactive Rebasing enables you to commit earlier and with more frequency, because you are able to edit your history before submitting your pull request
- To use Interactive Rebasing, run
  - `git rebase -i <base-commit-hash>`
  - A base commit hash can be a commit hash or HEAD~X where X stands for how many commits you want to go back (therefore a base commit)

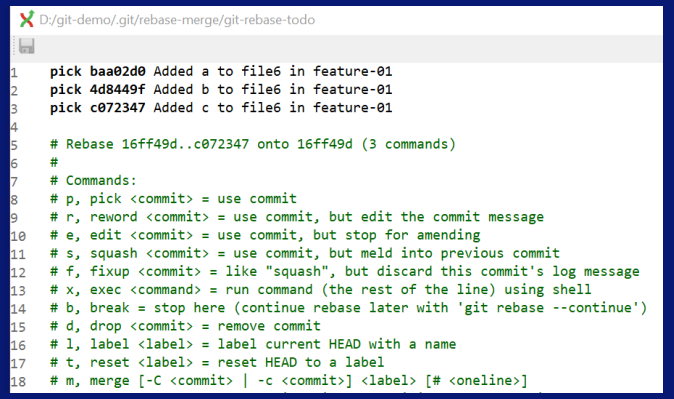
Interactive Rebasing

- Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch
- It can alter branch's commit history to clean up its messy history before merging a feature branch into master
- Sometimes a feature implementation can have multiple commits; including changes, typo corrections, temporary commits etc
- But In order to keep your master commits log nice and clean, it's important to have one commit per feature (or bug fix). You can have 5, 10 or 100 commits in your feature branch, but once it's ready to be merged into master, it's highly recommended to squash those commits into a nice clean single commit
- When running an interactive rebase with `rebase -i`, there are mainly three actions we may want to perform in a commit:
  1. `pick: pick a commit.`
  2. `squash: squash this commit into the previous one.`
  3. `drop: drop this commit altogether.`
- Interactive Rebasing enables you to commit earlier and with more frequency, because you are able to edit your history before submitting your pull request
- To use Interactive Rebasing, run
  - `git rebase -i <base-commit-hash>`
  - A base commit hash can be a commit hash or HEAD~X where X stands for how many commits you want to go back (therefore a base commit)

Git

Interactive Rebasing

- Switch to feature-01 branch
  - `git checkout feature-01`
- Modify file6.txt with minor changes and commit it many times as shown in the image
  - `echo "a=1" >> file6.txt`
  - `git commit -am "Added a to file6 in feature-01"`
- Check the commit log to see the latest commits
  - `git log --one-line`
- Notice that there are multiple commits that are not meaningful enough to keep them around in commit history
- Let's squash all these commits into a single commit
- Run the below command to open interactive rebasing for last 3 commits. This open a text editor to choose the commits
  - `git rebase -i HEAD~3`




```
428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git status
On branch feature-01
nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ ls -a
./  ../  .git/  .gitignore  file1.txt  file2.txt
file3.txt  file4.txt  file5.txt  file6.txt  TODO.txt

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ cat file6.txt
My Sixth File
Modified file6 in feature-01

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ echo "a=1" >> file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git commit -am "Added a to file6 in feature-01"

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ echo "b=1" >> file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git commit -am "Added b to file6 in feature-01"

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ echo "c=1" >> file6.txt

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git commit -am "Added c to file6 in feature-01"

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ cat file6.txt
My Sixth File
Modified file6 in feature-01
a=1
b=1
c=1

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
c072347 (HEAD -> feature-01) Added c to file6 in feature-
01
4d8449f Added b to file6 in feature-01
baa02d0 Added a to file6 in feature-01
16ff49d (origin/feature-01) Modified file6 in feature-01
de82d83 (origin/master, master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```

Git

Interactive Rebasing

- From the editor we get to choose the commits to stay, commits to be squashed and many more options
- Let's pick a commit and squash the other two to it
- Edit as shown in the image, save the file and exit
- Give a commit message and save it

D:/git-demo/.git/rebase-merge/git-rebase-todo

```
pick baa02d0 Added a to file6 in feature-01
pick 4d8449f Added b to file6 in feature-01
pick c072347 Added c to file6 in feature-01

# Rebase 16ff49d..c072347 onto 16ff49d (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

D:/git-demo/.git/rebase-merge/git-rebase-todo

```
pick baa02d0 Added a to file6 in feature-01
squash 4d8449f Added b to file6 in feature-01
squash c072347 Added c to file6 in feature-01

# Rebase 16ff49d..c072347 onto 16ff49d (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

commit message

D:/git-demo/.git/COMMIT_EDITMSG

```
This is a combination of 3 commits.
# This is the 1st commit message:

# Added a to file6 in feature-01

# This is the commit message #2:

# Added b to file6 in feature-01

# This is the commit message #3:

# Added c to file6 in feature-01

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Feb 13 17:56:28 2021 +0530
#
# interactive rebase in progress; onto 16ff49d
# Last commands done (3 commands done):
#    squash 4d8449f Added b to file6 in feature-01
#    squash c072347 Added c to file6 in feature-01
```

Git

Interactive Rebasing

- As seen from the log, it is clear that interactive rebase eliminated insignificant commits and made feature branch's history much easier to understand. This is something that git merge simply cannot do
  - `git log --oneline`
- Once done, push your changes to the remote
  - `git push origin feature-01`

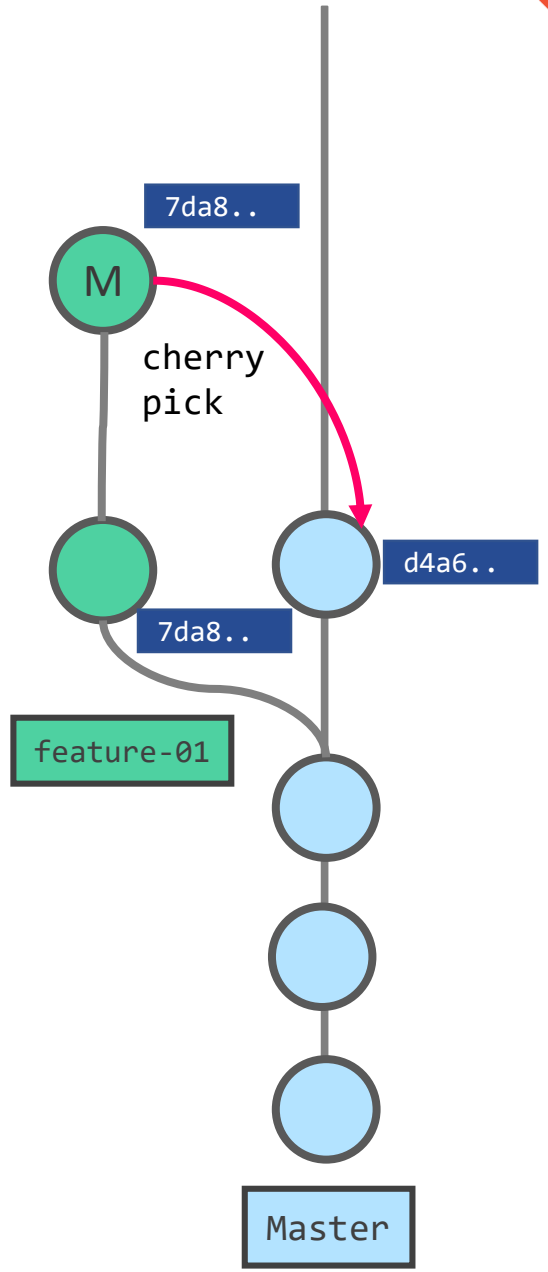
- Never use git rebase on public branches
- Git Rebase, Git Interactive rebase re-writes history by creating new commits, resulting in a new hash code for each one of those commits.
- If you try to push the rebased master branch back to a remote repository, Git will prevent you from doing so because it conflicts with the remote master branch as you have changed the history with rebase
- You can still force push `using git push origin <branch> -f`

```
428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ cat file6.txt
My Sixth File
Modified file6 in feature-01
a=1
b=1
c=1

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
c072347 (HEAD -> feature-01) Added c to file6 in feature-01
4d8449f Added b to file6 in feature-01
baa02d0 Added a to file6 in feature-01
16ff49d (origin/feature-01) Modified file6 in feature-01
de82d83 (origin/master, master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git rebase -i HEAD~3
[detached HEAD c61300f] This is a combination of 3 commits.
 Date: Sat Feb 13 17:56:28 2021 +0530
 1 file changed, 3 insertions(+)
Successfully rebased and updated refs/heads/feature-01.

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
c61300f (HEAD -> feature-01) This is a combination of 3 commits.
16ff49d (origin/feature-01) Modified file6 in feature-01
de82d83 (origin/master, master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```

Git

Cherry Pick

- Cherry picking is the act of picking a commit from a branch and applying it to another
- `git cherry-pick` can be useful for undoing changes. For example, say a commit is accidently made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong
- It is useful in many scenarios like:
  - Let's say you've been working on a feature branch and decide that most of the work is not good
  - You want to abandon the branch, but there are one or more commits that are good
  - You don't want to lose that work, so you can cherry-pick the specific commits that are good, applying them to a different/new branch

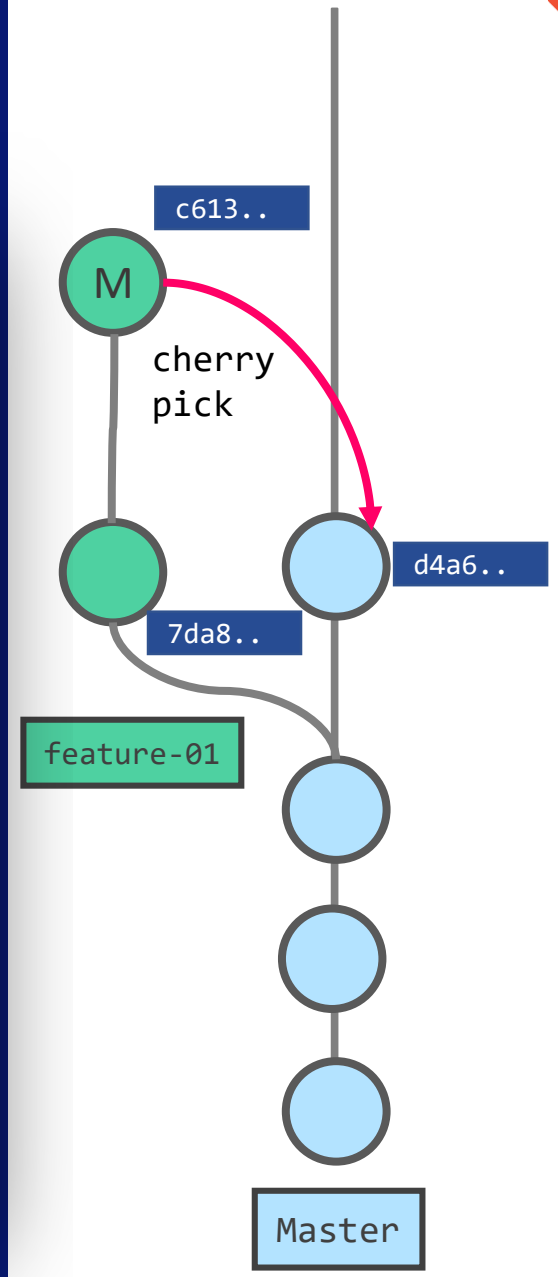


Git

Cherry Pick

- From the earlier demo we have file6.txt with updated information in feature-01 branch. Run git log to see the commit hash
  - `git log --one-line`
- Lets pull this commit onto master branch. Switch to master branch and apply cherry pick with commit id of file6 from feature-01 branch
  - `git checkout master`
  - `git cherry-pick c61300f`
- There will be merge conflicts as expected and git will show the conflicted files
- Edit those files, save and quit
- Add them to staging to continue with cherry picking. `git add .`
- Run below command to proceed with cherry picking. Git will prompt for commit message
  - `git cherry-pick --continue`
- Once done, a new commit hash will be generated.

```
428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ cat file6.txt
My Sixth File
Modified file6 in feature-01
a=1
b=1
c=1

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git log --oneline
c61300f (HEAD -> feature-01, origin/feature-01) This is a
combination of 3 commits.
16ff49d Modified file6 in feature-01
de82d83 (origin/master, master) Update file5.txt
....

428991@LINL190904638 MINGW64 /d/git-demo (feature-01)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file6.txt
My Sixth File

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git cherry-pick c61300f
Auto-merging file6.txt
CONFLICT (content): Merge conflict in file6.txt
error: could not apply c61300f... This is a combination of 3
commits.
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
```



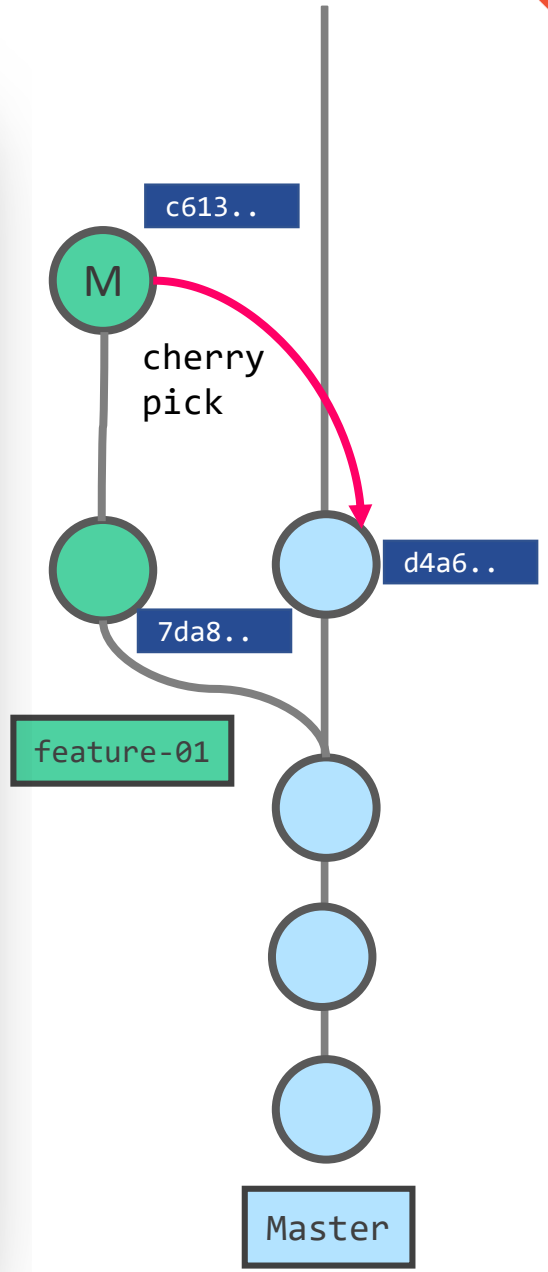
Git

Cherry Pick

Some Cherry-pick commands

- fix conflicts and run:
  - git: cherry-pick --continue"
- Skip the patch:
  - git cherry-pick --skip
- Abort cherry-pick:
  - git cherry-pick --abort

```
428991@LINL190904638 MINGW64 /d/git-demo (master|CHERRY-PICKING)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

You are currently cherry-picking commit c61300f.
  (fix conflicts and run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick
operation)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file6.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master|CHERRY-PICKING)
$ git add .

428991@LINL190904638 MINGW64 /d/git-demo (master|CHERRY-PICKING)
$ git cherry-pick --continue
[master a6ecdfc] This is a combination of 3 commits.
 Date: Sat Feb 13 17:56:28 2021 +0530
 1 file changed, 5 insertions(+)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file6.txt
My Sixth File
Modified file6 in feature-01
a=1
b=1
c=1


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
a6ecdfc (HEAD -> master) This is a combination of 3 commits.
de82d83 (origin/master) Update file5.txt
e126282 Update file4.txt
7f953dd Added Sixth File to Dev Branch
755a2fc Added .gitignore
2fedc59 Second Commit, Added file4.txt and file5.txt
53b6941 Initial Commit
```



Git

Undoing Changes: Checkout, Revert and Reset

- If you have made changes to a file that you don't like, and they have not been committed yet, to undo those changes run
  - `git checkout <file-name>`
- If changes are committed and a new hash is generated, to undo run
  - `git revert 2f5451f --no-edit`
  - This will make a new commit that is the opposite of the existing commit, reverting the file(s) to their previous state as if it was never changed.
  - `--no-edit` option prevents git from asking you to enter in a commit message. If you don't add that option, you'll end up in the VIM text editor where a commit message is to be added
- Run git log to see a new commit with reverted changes
  - `git log --one-line`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "a=1" >> file4.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main
a=1

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git checkout file4.txt
Updated 1 path from the index

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "b=1" >> file4.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main
b=1

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git commit -am "Modified file4.txt"
warning: LF will be replaced by CRLF in file4.txt.
The file will have its original line endings in your
working directory
[master e52643f] Modified file4.txt
 1 file changed, 1 insertion(+)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
e52643f (HEAD -> master) Modified file4.txt
a6ecdfc This is a combination of 3 commits.
de82d83 (origin/master) Update file5.txt
...


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git revert e52643f --no-edit
[master e406f70] Revert "Modified file4.txt"
 Date: Sat Feb 13 19:40:33 2021 +0530
 1 file changed, 1 deletion(-)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main
```

Git

Undoing Changes: Checkout, Revert and Reset

- Undo your last commit that is not pushed yet
  - `git reset --soft HEAD~1`
  - `git reset --hard HEAD~1`
  - `--soft will undo the commit but also retains the changes so that we can edit and commit them again. Changes appear as uncommitted`
  - `--hard will completely remove those changes`
- To undo last N commits
  - `git reset --soft HEAD~N`
  - `git reset --soft HEAD~5`
- We can also use commit hash to undo the changes
  - `git reset --soft <commit-hash>`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "b=1" >> file4.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git commit -am "Modified file4.txt"

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git log --oneline
e4152a3 (HEAD -> master) Modified file4.txt
de82d83 (origin/master) Update file5.txt
...

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git reset --soft HEAD~1

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file4.txt


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ cat file4.txt
My Fourth File
Modified Fourth File in Main
b=1
```

Git

Git Stash

- Consider the scenario you are working on a `feature-01` branch to implement a new feature
- You have made lot of changes to the code in that branch but they are not yet ready to be committed
- Now, you received a request to fix a bug in the `master` branch and you are to switch to `master` branch to fix the issue
- But, you don't want to lose your work in the `feature-01` branch
- In this case, `git stash` command will allow you to save your local modifications and revert back to the working directory that is in line with the most recent HEAD commit
- In the sense, `git stash` takes the present state of the working file and puts in on the stack and gives you back a clean working file. Now you are free to switch to other branch
- So `git stash` lets us preserving the changes and restore them whenever needed
- `git stash apply` will restore all the saved changes to the working area again
- When you are done with the stashed element or want to delete it from the directory, run the `git stash drop` command. It will delete the last added stash item by default

Git

Git Stash

- Git stash with message
  - `git stash save “Your stash message”`
- Stash untracked files as well
  - `git stash save --include-untracked`
- List all stashes
  - `git stash list`
- Get summary of the stash diffs
  - `git stash show -p`
  - `git stash show stash@{0}`
- Apply top most stash in the stack(replace 0 with other integer to apply other stash)
  - `git stash apply stash@{0}`
- Apply the top stash and delete it as well. Use index 1 for second stash
  - `git stash pop stash@{0}` (or)
  - `git stash pop`
- Delete a specific stash
  - `git stash drop stash@{0}`
- Delete all stashes
  - `git stash clear`
- Create a branch and apply the stash
  - `git stash branch <name> stash@{1}`

```
428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "b=1" >> file4.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ echo "a=1" >> file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git add file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
        modified:   file5.txt

Changes not staged for commit:
        modified:   file4.txt


428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git stash save "feature implementation"

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git stash list
stash@{0}: On master: feature implementation

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git stash show stash@{0}
 file4.txt | 1 +
 file5.txt | 1 +
 2 files changed, 2 insertions(+)

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git stash pop
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
        modified:   file4.txt
        modified:   file5.txt

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
        modified:   file4.txt
        modified:   file5.txt

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

428991@LINL190904638 MINGW64 /d/git-demo (master)
$ git stash list
```

Git

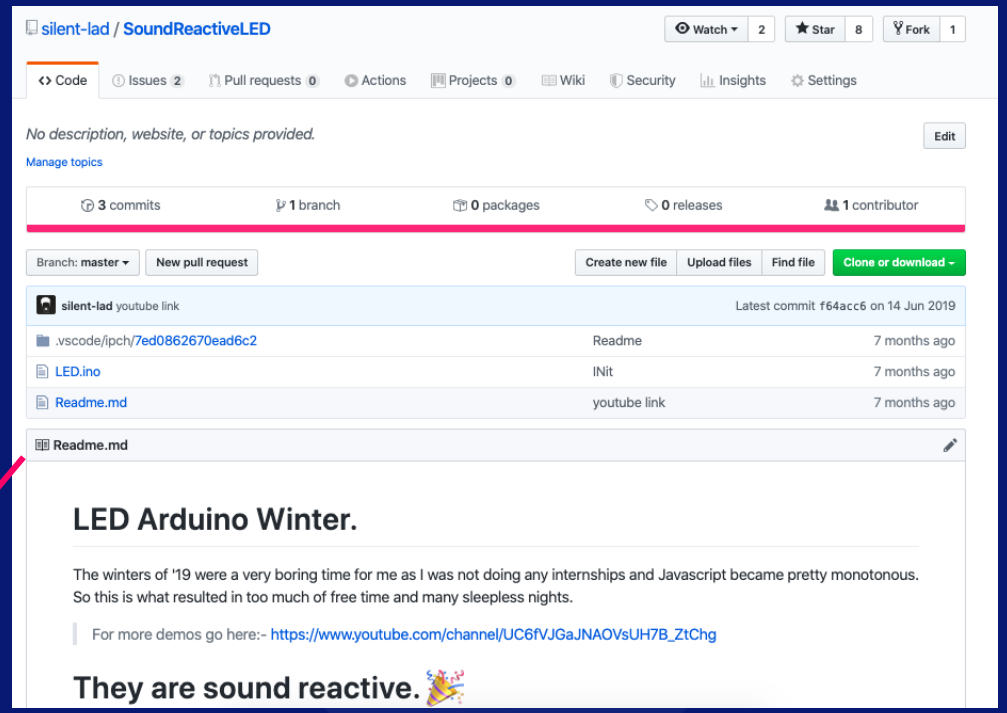
Recover deleted data: Git Reflog

- The `reflog` is a special mechanism that contains any changes to the data in your repository
- This includes committing changes, creating and checking out branches, and even hard resets
- It's a handy timeline, going a step further from the usual history tree, that you can use to recover any data you may have deleted or misplaced
- More about reflog here https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog

Git

ReadMe File

- A ReadMe file is a standard place for instructions or documentation that you want to share with people about a repo
- Every repository should have a README.md file which is a MarkDown file
- Create a file named README.md in the root (based) folder of the Git repo
- Use Markdown to format headings, lists, links, images etc.
- Here are some guides for the Markdown syntax:
  - https://github.com/tchapi/markdown-cheatsheet/blob/master/README.md
- Live MarkDown syntax rendering
  - https://stackedit.io/app#



This README.md shows instructions on how to interface LED to Arduino and load the .ino file to Arduino IDE and to upload it to the board

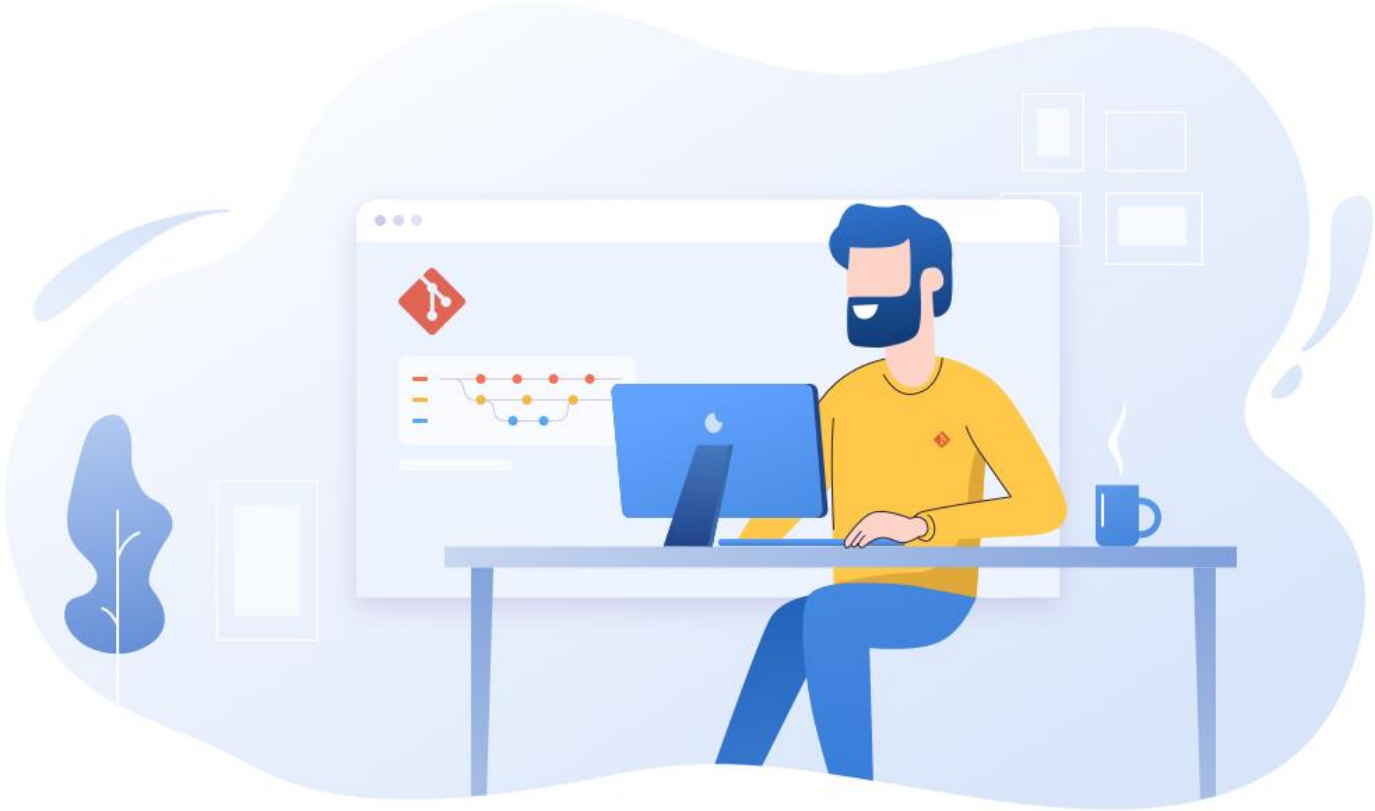
Git

Logs

- `git log` shows the commit log
- `git log origin/master` shows the commit log
- `git log --patch -2` :shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries
- `git log --stat` shows abbreviated stats for each commit
- `git log --pretty=oneline` (or) `git log --oneline` :--pretty changes the log output to formats other than the default. `--oneline` value for this option prints each commit on a single line
- `git log --name-only` – shows only files changed
- `git show`
- `git show <commit-id>`
- `git show --pretty="format:" --name-only <commit-id>`
- `git log --decorate`
- `git log --graph --decorate`
- `git log --follow file1.txt`: Show the commits that changed a particular file
- `git log a-branch..b-branch` or `git log dev..master` :Show the commits that are on one branch and not on the other. This will show commits on a-branch that are not on b-branch.

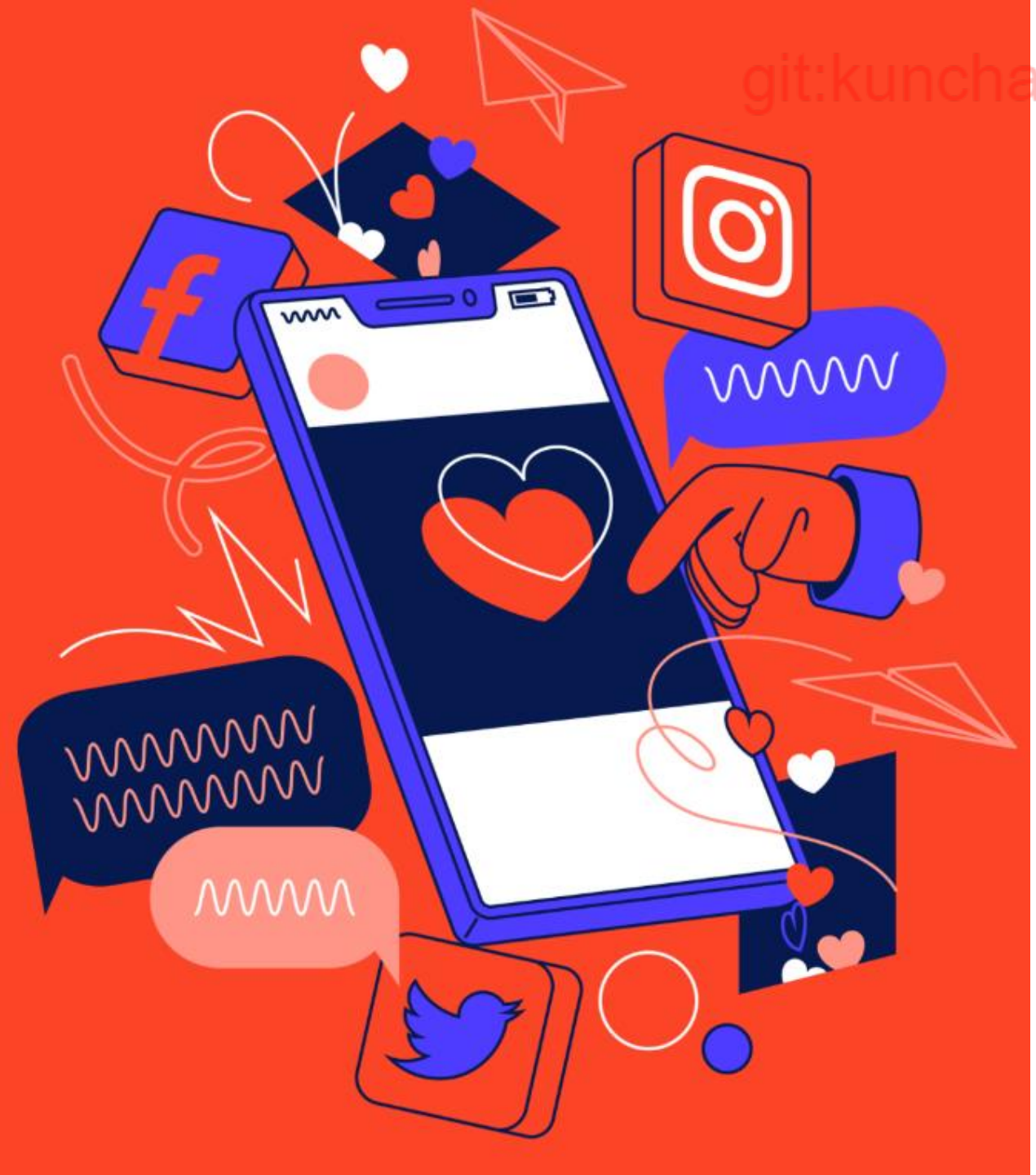
References

- https://git-scm.com/book/en/v2
- https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified
- https://mirrors.edge.kernel.org/pub/software/scm/git/docs/gittutorial.html
- http://gitready.com/



We have now reached the end of this tutorial!

Do like and join our fb group
for more tutorials like this.



Like our good work and want to sponsor/contribute for more tutorials like this? Send an email to kunchalavikram1427@gmail.com
Your support is highly appreciated!

Subscribe to my Facebook page:
https://www.facebook.com/vikram.devops
and join my group:
https://www.facebook.com/groups/171043094400359
for all latest updates on DevOps, Python and IoT

https://www.youtube.com/channel/UCE1cGZfooxT7-VbqVbuKjMg

/kunchalavikram1427

Thank You