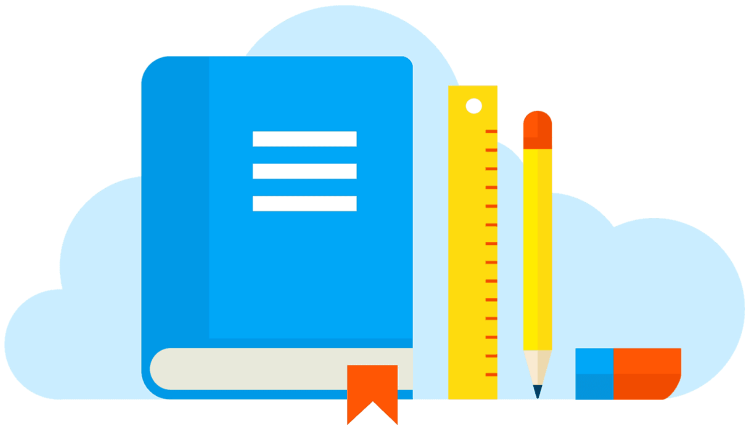
Girl Script Foundation- Introduction to Git and Github



Summary

Date: 23/04/2020

Introduction to Git and GitHub Day 4

- Git Log
- Git tagging
- Comparing Commits
- Parallel development in Repository
- Branching
- Merging
- Rebasing

1) Git Log

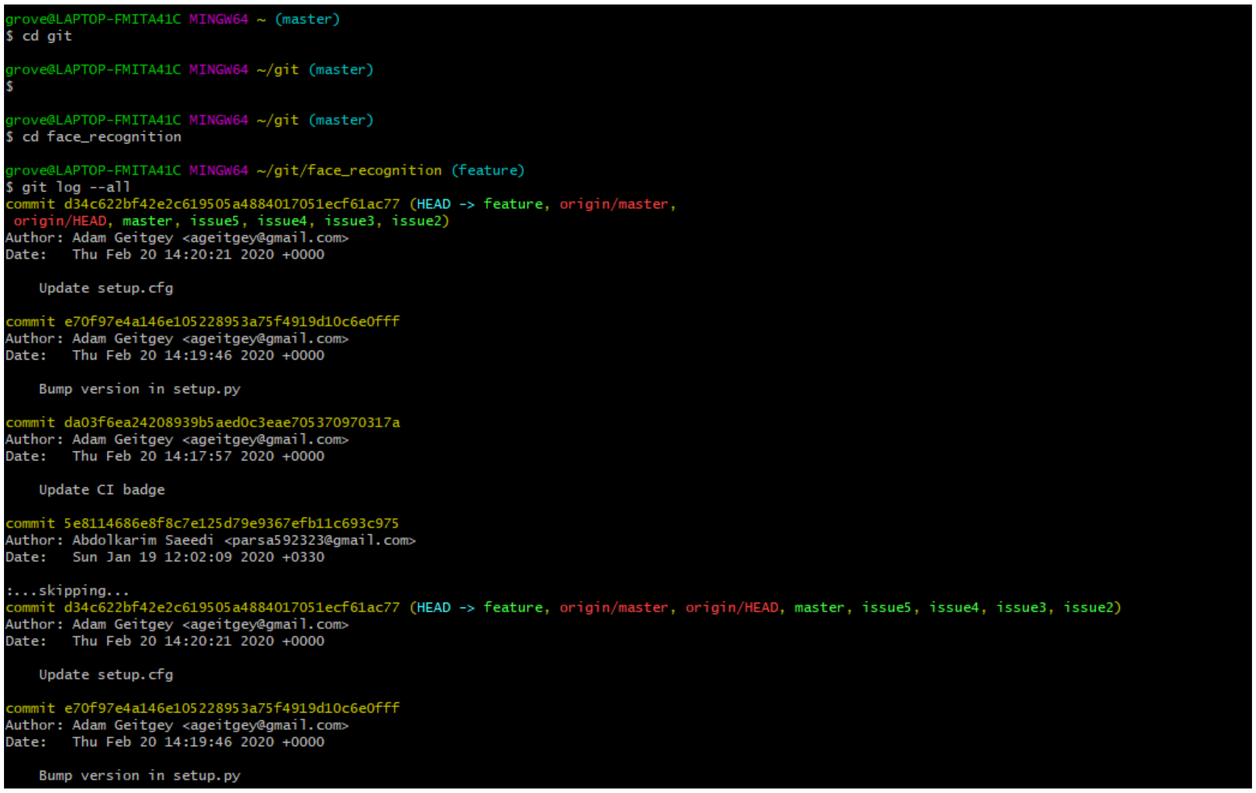
The simplest version of the log command shows the commits that lead up to the state of the currently checked out branch.

These commits are shown in reverse chronological order (the most recent commits first).

https://git-scm.com/docs/git-log link to official documentation.

1) git log --all

   You can force the log tool to display all commits (regardless of the branch checked out) by using the –all option.

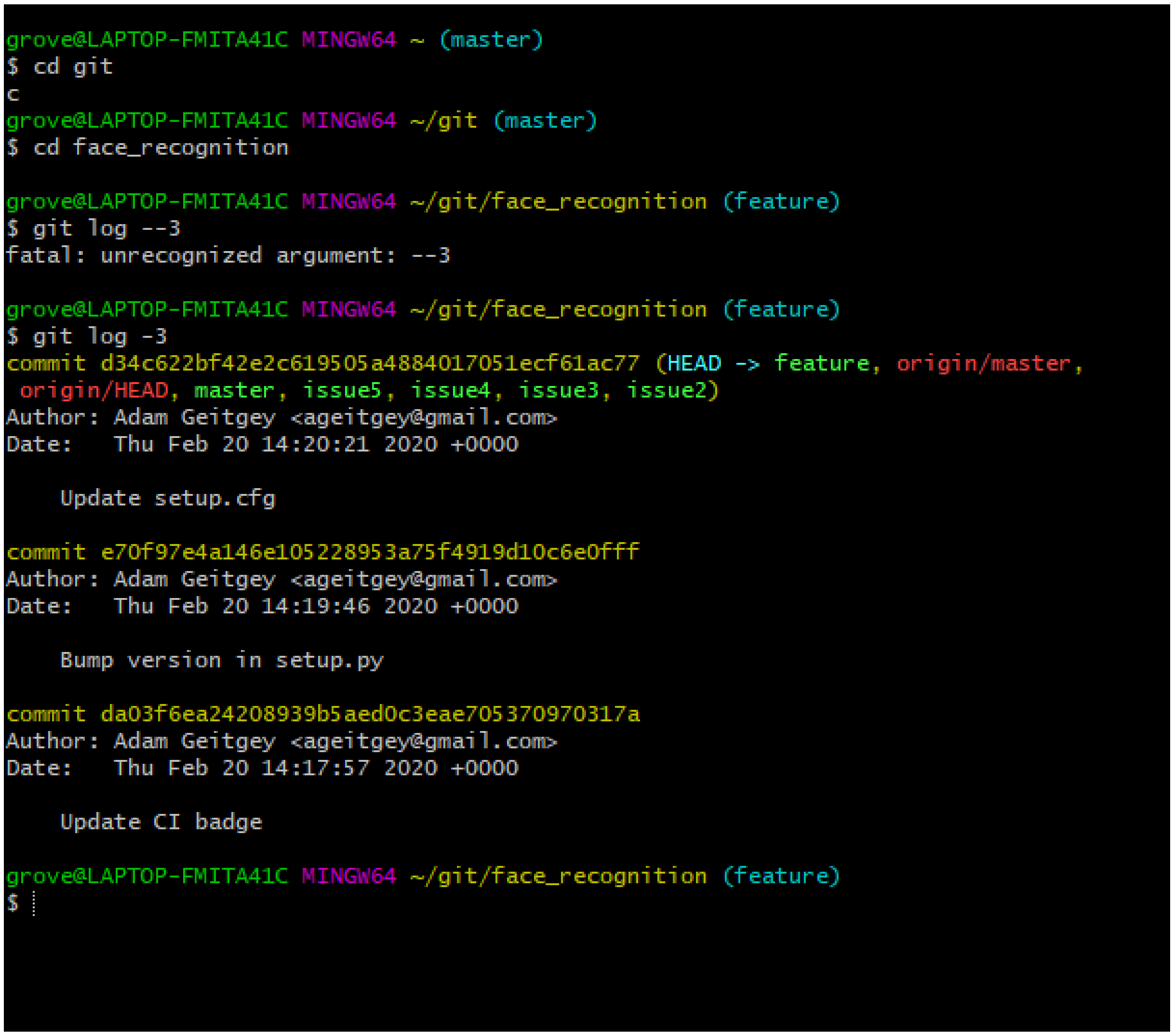


```
grove@LAPTOP-FMITA41C MINGW64 ~ (master)
$ cd git

grove@LAPTOP-FMITA41C MINGW64 ~/git (master)
$

grove@LAPTOP-FMITA41C MINGW64 ~/git (master)
$ cd face_recognition

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git log --all
commit d34c622bf42e2c619505a4884017051ecf61ac77 (HEAD -> feature, origin/master,
 origin/HEAD, master, issue5, issue4, issue3, issue2)
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:20:21 2020 +0000

    Update setup.cfg

commit e70f97e4a146e105228953a75f4919d10c6e0fff
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:19:46 2020 +0000

    Bump version in setup.py

commit da03f6ea24208939b5aed0c3eae705370970317a
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:17:57 2020 +0000

    Update CI badge

commit 5e8114686e8f8c7e125d79e9367efb11c693c975
Author: Abdolkarim Saeedi <parsa592323@gmail.com>
Date:   Sun Jan 19 12:02:09 2020 +0330

:...skipping...
commit d34c622bf42e2c619505a4884017051ecf61ac77 (HEAD -> feature, origin/master, origin/HEAD, master, issue5, issue4, issue3, issue2)
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:20:21 2020 +0000

    Update setup.cfg

commit e70f97e4a146e105228953a75f4919d10c6e0fff
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:19:46 2020 +0000

    Bump version in setup.py
```

2) git log -n

View n Most Recent Commits

The most trivial way to filter commits is to limit output to the 'n' most recently committed ones.



```
grove@LAPTOP-FMITA41C MINGW64 ~ (master)
$ cd git
c
grove@LAPTOP-FMITA41C MINGW64 ~/git (master)
$ cd face_recognition

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git log --3
fatal: unrecognized argument: --3

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git log -3
commit d34c622bf42e2c619505a4884017051ecf61ac77 (HEAD -> feature, origin/master,
 origin/HEAD, master, issue5, issue4, issue3, issue2)
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:20:21 2020 +0000

    Update setup.cfg

commit e70f97e4a146e105228953a75f4919d10c6e0fff
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:19:46 2020 +0000

    Bump version in setup.py

commit da03f6ea24208939b5aed0c3eae705370970317a
Author: Adam Geitgey <ageitgey@gmail.com>
Date:   Thu Feb 20 14:17:57 2020 +0000

    Update CI badge

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$
```

Git Tagging

Official documentation -

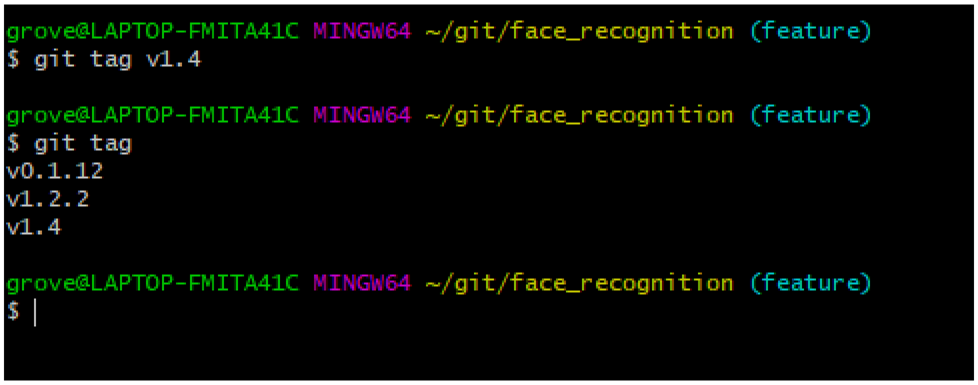
git-scm.com › book › Git-Basics-Tagging

1) Create Tag

   git tag <tagname>

2) To list the tags

   git tag



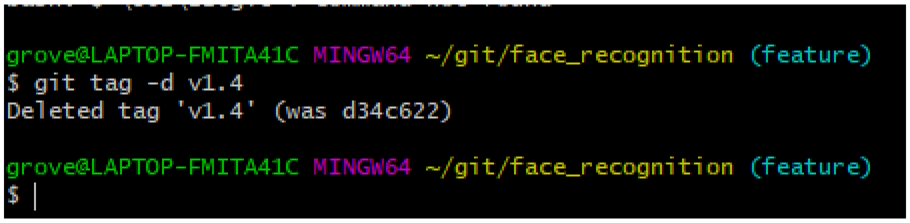
```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git tag v1.4

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git tag
v0.1.12
v1.2.2
v1.4

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ |
```

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release.

3) git tag -d <tagname>

Deleting Tags

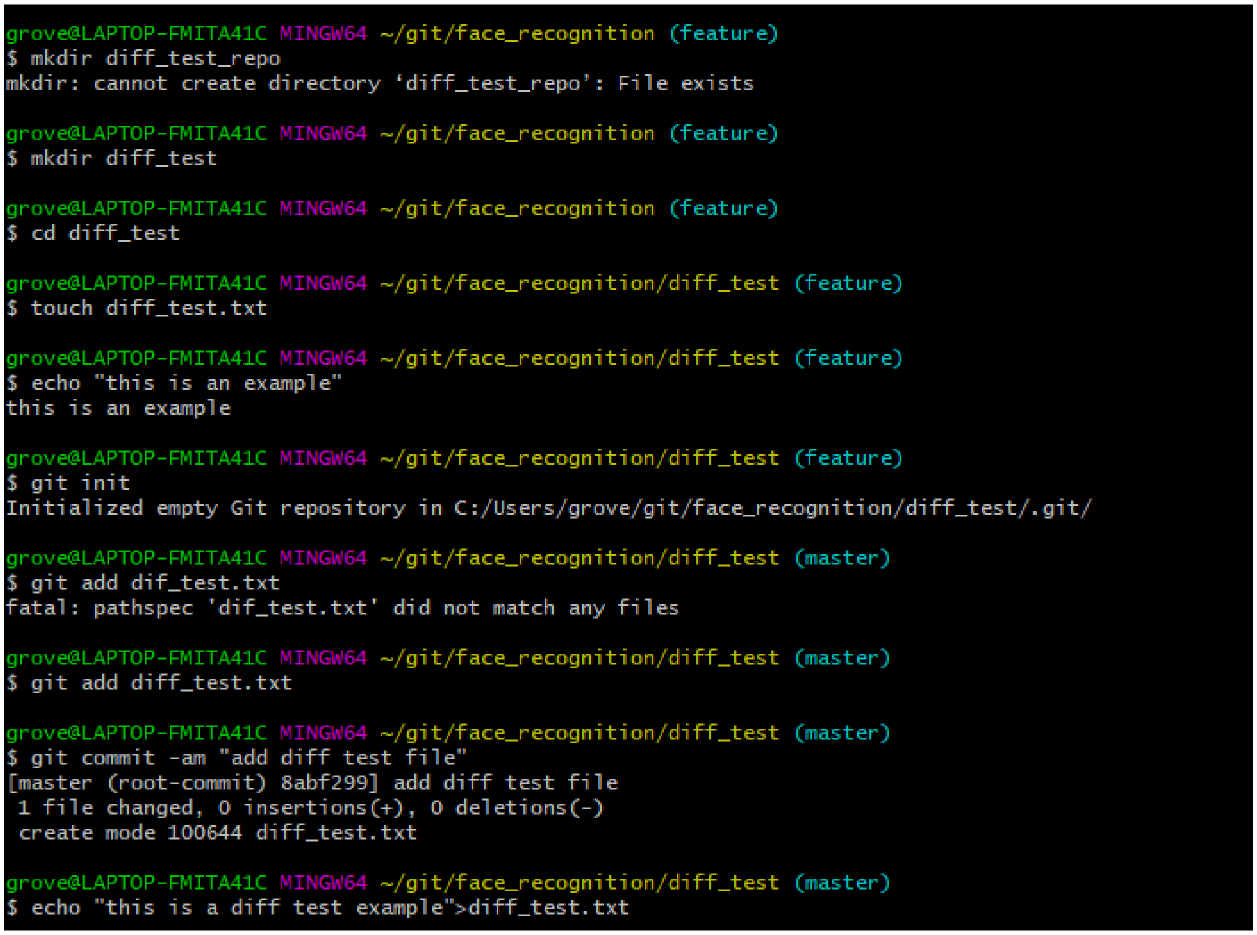
Deleting tags is a straightforward operation. Passing the -d option and a tag identifier to git tag will delete the identified tag.



```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git tag -d v1.4
Deleted tag 'v1.4' (was d34c622)

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ |
```

Comparing Commits

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more.



```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ mkdir diff_test_repo
mkdir: cannot create directory ‘diff_test_repo’: File exists

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ mkdir diff_test

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ cd diff_test

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (feature)
$ touch diff_test.txt

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (feature)
$ echo "this is an example"
this is an example

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (feature)
$ git init
Initialized empty Git repository in C:/Users/grove/git/face_recognition/diff_test/.git/

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git add dif_test.txt
fatal: pathspec 'dif_test.txt' did not match any files

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git add diff_test.txt

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git commit -am "add diff test file"
[master (root-commit) 8abf299] add diff test file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 diff_test.txt

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ echo "this is a diff test example">diff_test.txt
```

Parallel Development in Repository

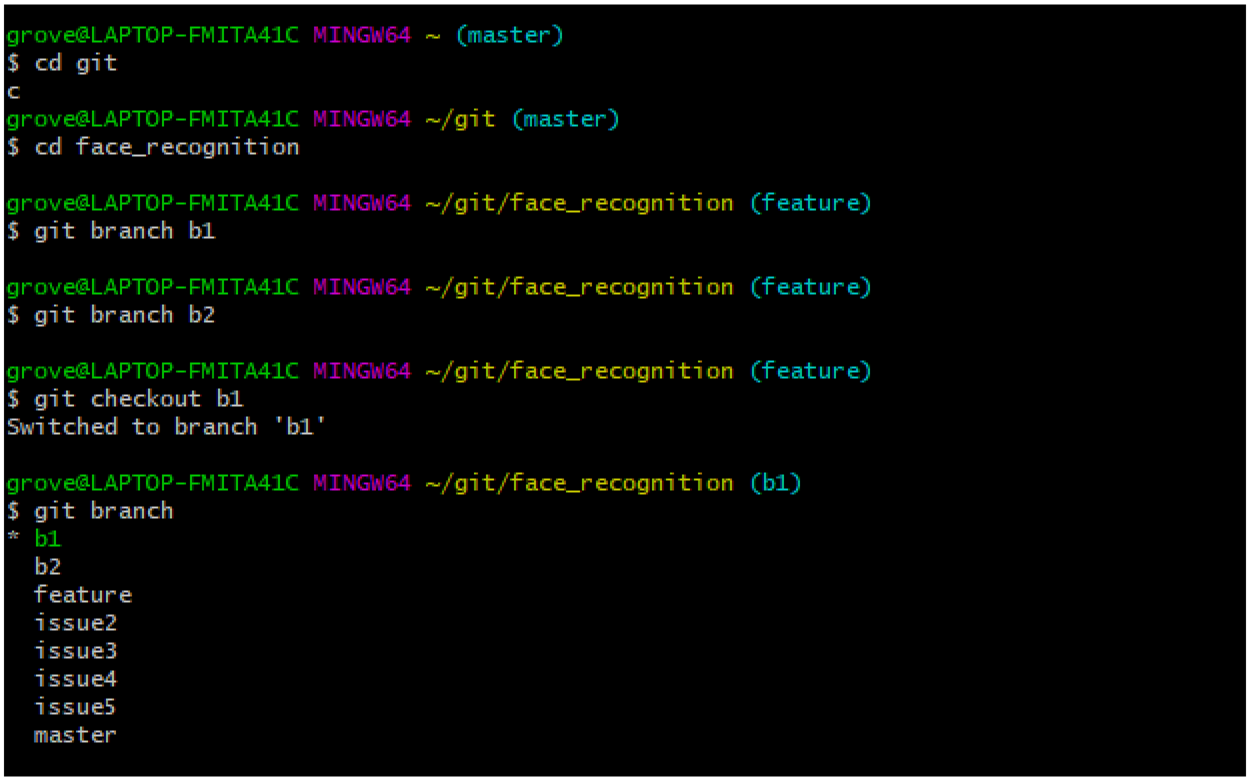
All the source will available in "central repository"
. I assume 3 developers working in parallel on the same project hosted on the 'central repository'.

- DevA clones the central repository to his local machine.

- DevA checkout the master branch. So now the working set will have the latest from master branch.
- To work on a feature / bug, he creates a new branch, say feature_123. He should now be placed in the feature_123 branch
- He develops the feature, keep commit ‘ing the changes to feature_123 branch.
- Periodically he pull the changes from central repository to get the latest changes from other developers.
- Once done he push the changes to origin (i.e., central repository).

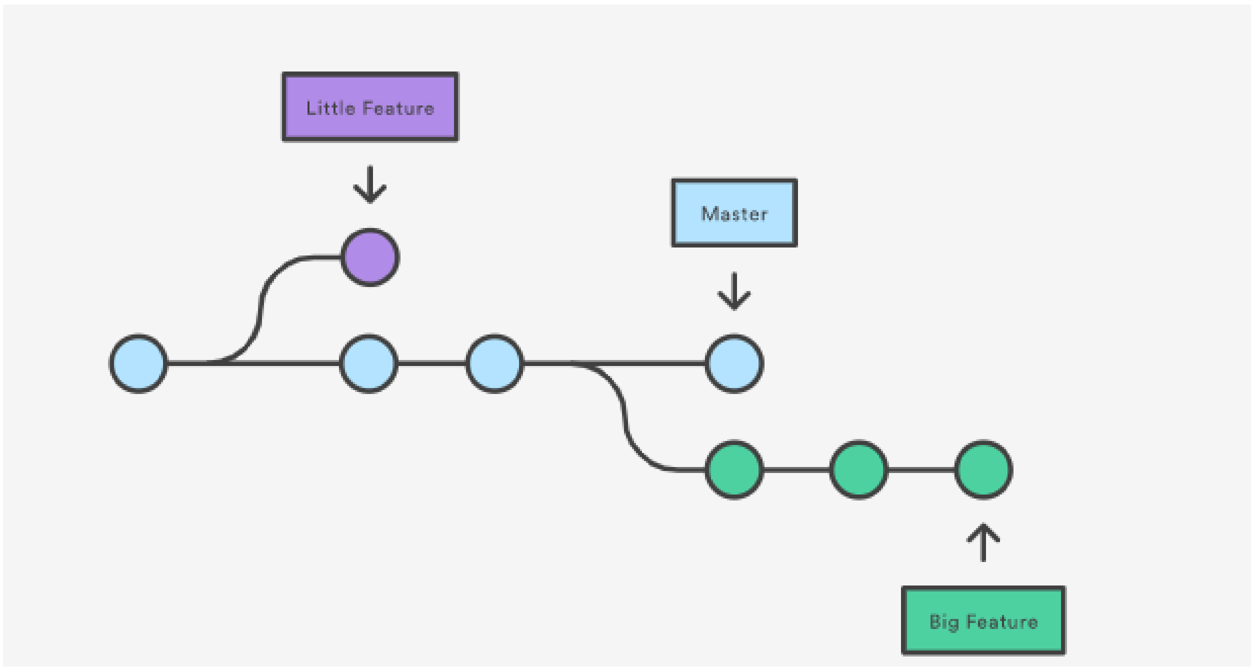
The Developers (DevA, B, and C) is not allowed to merge their branches into the master to avoid future conflicts and confusions. The administrator of the repository should merge the feature branches to master.
For a simpler process after every merge of a feature_branch to master, all the developers should pull the changes from the central repository to refresh their local repositories. This restriction however would become bottleneck if the project is shared among 3+ developers.

Branching



```
grove@LAPTOP-FMITA41C MINGW64 ~ (master)
$ cd git
c
grove@LAPTOP-FMITA41C MINGW64 ~/git (master)
$ cd face_recognition

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git branch b1

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git branch b2

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (feature)
$ git checkout b1
Switched to branch 'b1'

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition (b1)
$ git branch
* b1
  b2
  feature
  issue2
  issue3
  issue4
  issue5
  master
```

Branching allows us to work in multiple parallel workspaces.

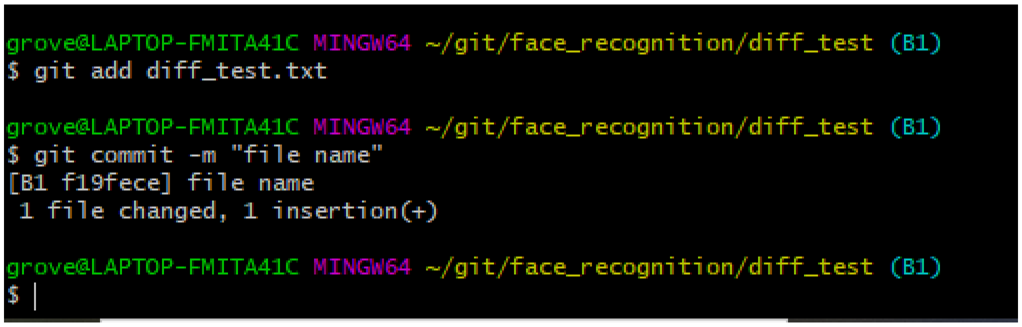


Git branches are effectively a pointer to a snapshot of your changes.

When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes.

This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

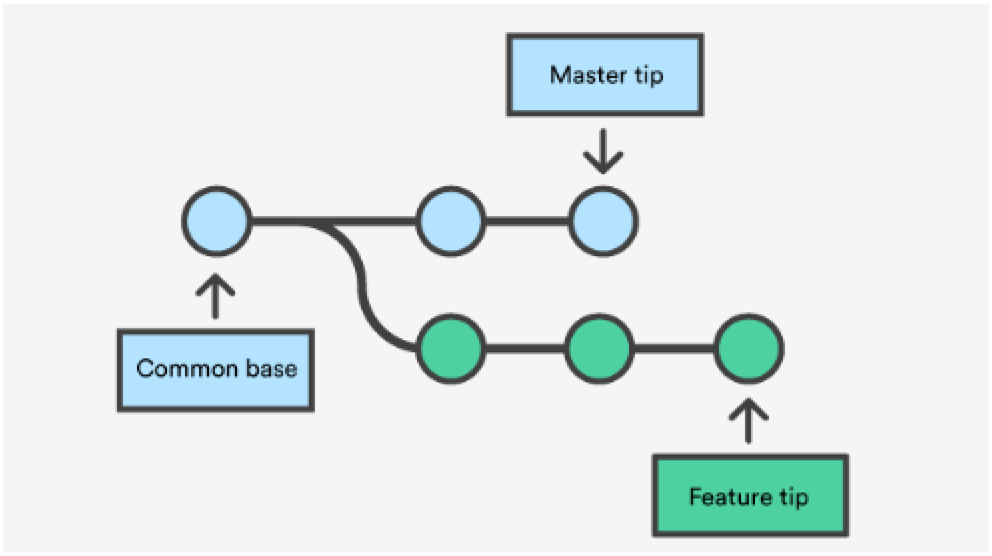


```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (B1)
$ git add diff_test.txt

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (B1)
$ git commit -m "file name"
[B1 f19fece] file name
 1 file changed, 1 insertion(+)

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (B1)
$ |
```

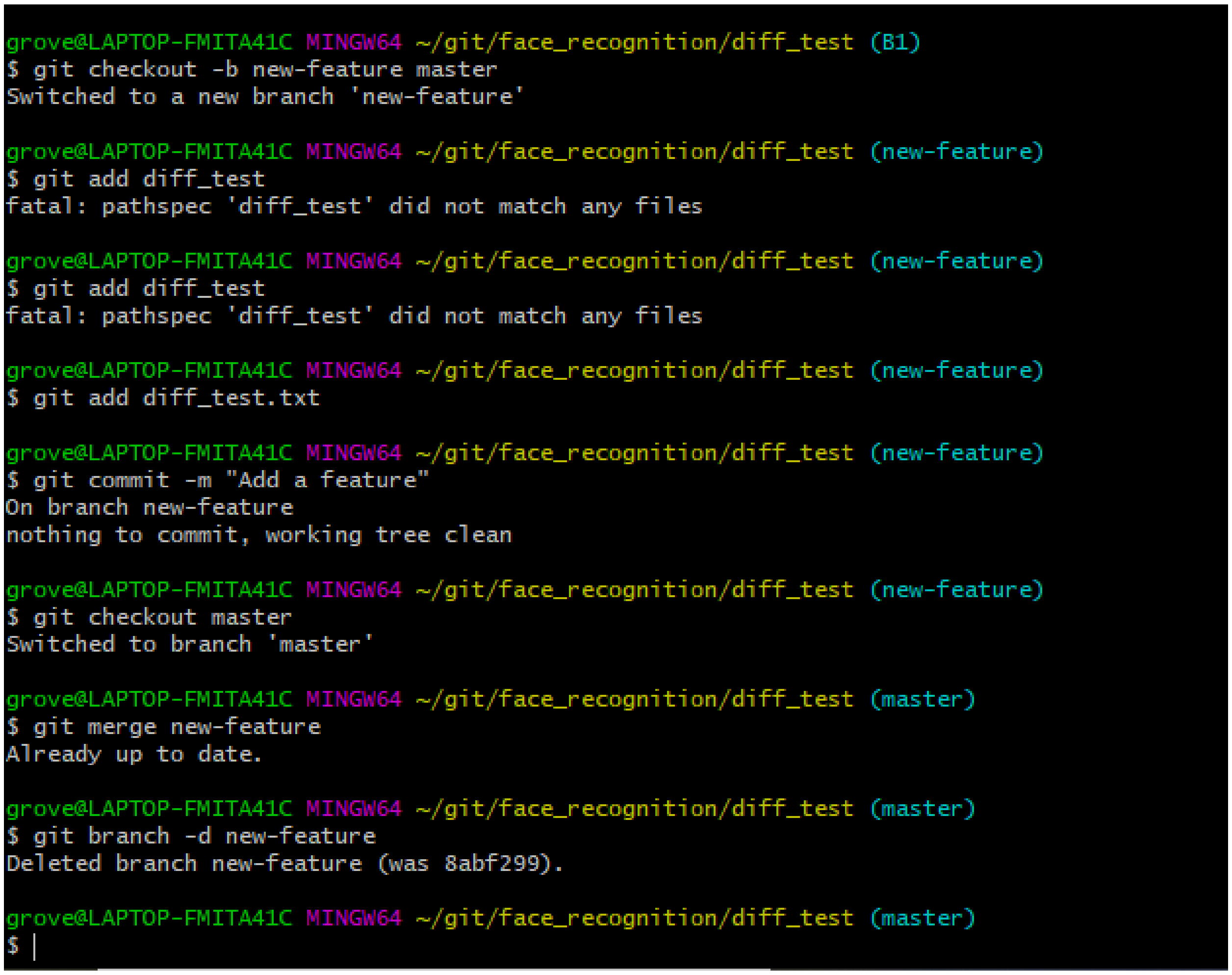
Git Merge

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence



The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.



```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (B1)
$ git checkout -b new-feature master
Switched to a new branch 'new-feature'

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (new-feature)
$ git add diff_test
fatal: pathspec 'diff_test' did not match any files

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (new-feature)
$ git add diff_test
fatal: pathspec 'diff_test' did not match any files

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (new-feature)
$ git add diff_test.txt

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (new-feature)
$ git commit -m "Add a feature"
On branch new-feature
nothing to commit, working tree clean

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (new-feature)
$ git checkout master
Switched to branch 'master'

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git merge new-feature
Already up to date.

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git branch -d new-feature
Deleted branch new-feature (was 8abf299).

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ |
```

Summary

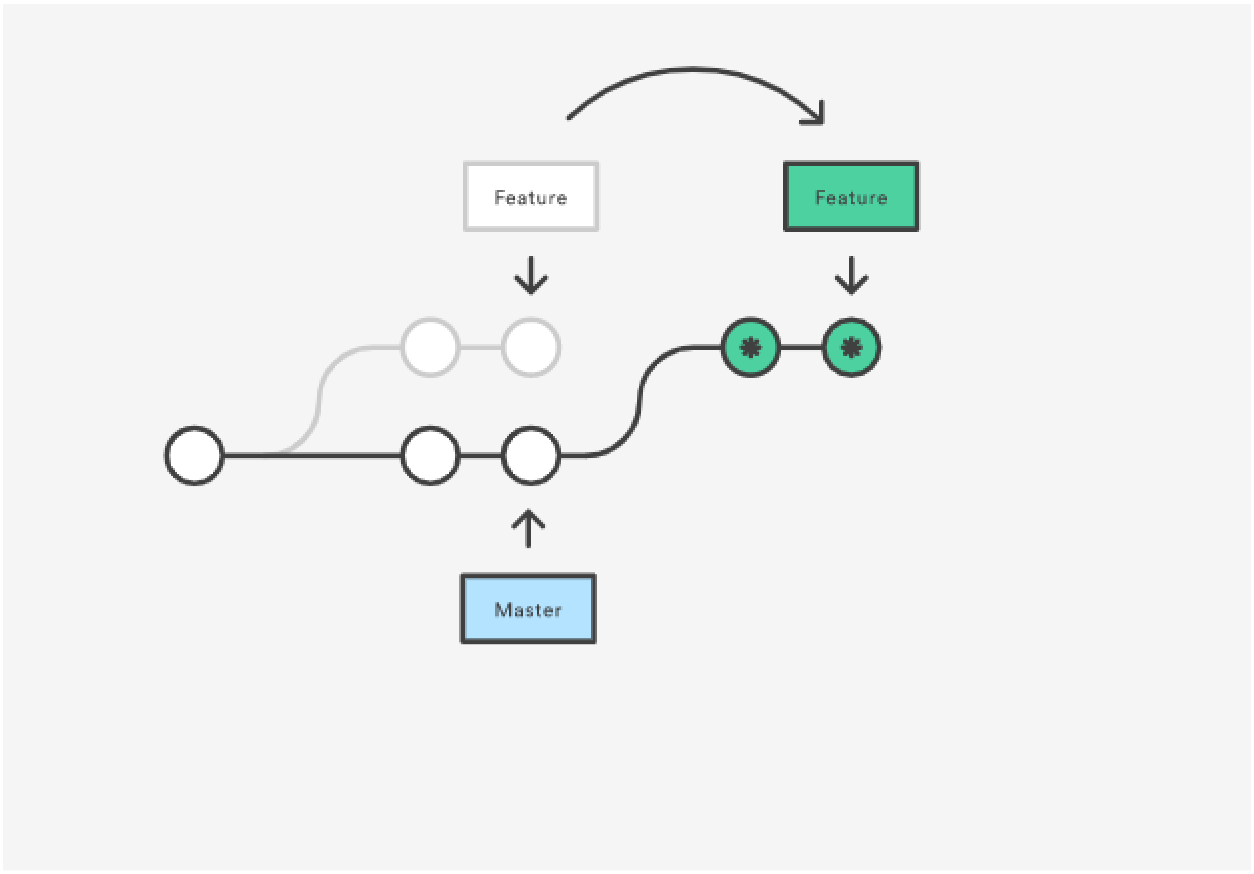
Some key take-aways are:

1. Git merging combines sequences of commits into one unified history of commits.
2. There are two main ways Git will merge: Fast Forward and Three way
3. Git can automatically merge commits unless there are changes that conflict in both commit sequences.

Git Rebase

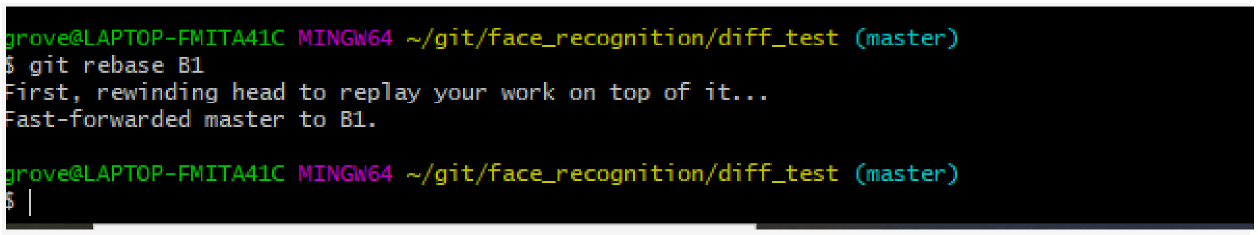
Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features.

ebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:

1. A bug is identified in the master branch. A feature that was working successfully is now broken.
2. A developer examines the history of the master branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
3. The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
4. Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.



```
grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ git rebase B1
First, rewinding head to replay your work on top of it...
Fast-forwarded master to B1.

grove@LAPTOP-FMITA41C MINGW64 ~/git/face_recognition/diff_test (master)
$ |
```