

Git Good

How to understand Git

Sumner Evans

26 September 2023

Mines ACM

My name is Sumner, I'm a **software engineer at Beeper**.

- I graduated from Colorado School of Mines in 2018 with my bachelor's in CS and 2019 with a master's in CS.
- I am an adjunct professor. Currently I'm teaching CSCI 406. I've taught 341, 400, and 564 in the past as well.
- I enjoy skiing, volleyball, and soccer.
- I'm a 4th degree black belt in ATA taekwondo.

Overview

1. Why use Git?
2. Commits
3. Branches
4. Merging
5. Rebasing
6. Remotes
7. Advanced Tips

This talk is interactive!

If you have questions at any point, feel free to interrupt me.

Why use Git?



Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? I

Example Scenario:

1. You start a project called "my-proj" and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of "my-proj" for backup purposes.
4. You continue development on "my-proj" but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn't have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *main()*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Version Control Systems (VCSs) such as Git solve these problems.

- VCS keeps track of *revisions*, changes in the code in entities called *changesets* or *commits*.
- Most VCS allow version merging. That means multiple people can be working on the same file and resolve discrepancies later. Git is very elegant in handling merge conflicts such as this.

Version Control Systems (VCSs) such as Git solve these problems.

- VCS keeps track of *revisions*, changes in the code in entities called *changesets* or *commits*.
- Most VCS allow version merging. That means multiple people can be working on the same file and resolve discrepancies later. Git is very elegant in handling merge conflicts such as this.

Git is popular

Git is a very popular version control system.

Services such as GitHub and GitLab provide free hosting for Git repositories.

It has become the de-facto industry standard for source control.

Git is a distributed version control system

Distributed because you can use it without being connected to a central server. You have a full copy of the code on your own computer.

Version control because it keeps track of changes to files.

But how does it keep track of all of the changes?

Git is a distributed version control system

Distributed because you can use it without being connected to a central server. You have a full copy of the code on your own computer.

Version control because it keeps track of changes to files.

But how does it keep track of all of the changes?

Commits



Commits: what are they?

Commits are sets of differences (diffs) in files.¹

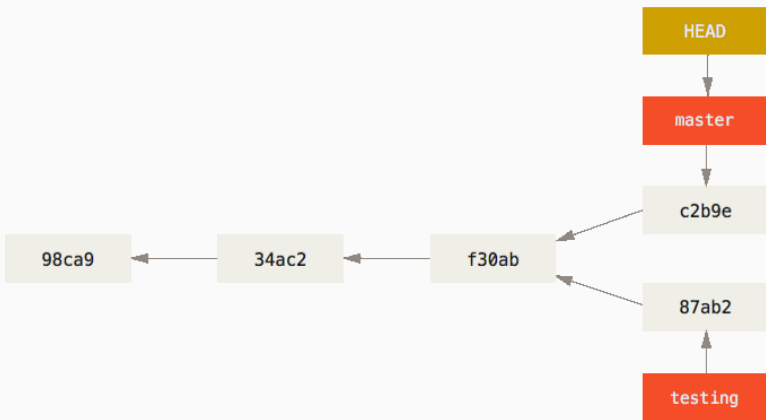
Commits reference their parent(s) and contain information about the changes made in the repo since that parent commit.

¹This is a bit of a lie, more on that later.

Commits: what are they?

Commits are sets of differences (diffs) in files.¹

Commits reference their parent(s) and contain information about the changes made in the repo since that parent commit.



¹This is a bit of a lie, more on that later.

Commits: they form a DAG, and are stored on the “heap”

Commits form a Directed Acyclic Graph (DAG). There are no loops in the graph, and every commit points to its parent(s).

Every single commit is stored in the `.git` directory of your repository² which can be thought of like a “heap” for commits. The parents of a commit are also stored in this “heap”.

We will return to this fact when we talk about *branches*.

²Technically another lie.

You can create a commit by running *git commit*. This will create a commit based on the checked-out commit with the changes that you have “staged”.

- You can stage changes by running *git add* and passing a list of files.
- You can stage all changes by running *git add -A*.
- Pro tip: If you want to add specific parts of files, use *git add -p*.

If you want to remove from the stage, you can run *git reset* (optionally passing a list of files to unstage).

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have “staged”.

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.
- **Pro tip:** If you want to add specific parts of files, use `git add -p`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have “staged”.

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.
- *Pro tip:* If you want to add specific parts of files, use `git add -p`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have “staged”.

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.
- **Pro tip:** If you want to add specific parts of files, use `git add -p`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have “staged”.

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.
- **Pro tip:** If you want to add specific parts of files, use `git add -p`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run **git status** to show the list of files staged for commit.

```
> git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

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

```
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:   git.pdf
    modified:   git.tex
```

The **git.tex** file has only some lines staged for commit.

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run **git status** to show the list of files staged for commit.

```
> git status
```

```
On branch master
```

```
Your branch is ahead of 'origin/master' by 1 commit.
```

```
(use "git push" to publish your local commits)
```

```
Changes to be committed:
```

```
(use "git restore --staged <file>..." to unstage)
```

```
    modified:   git.tex
```

```
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:   git.pdf
```

```
    modified:   git.tex
```

The **git.tex** file has only some lines staged for commit.

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run **git status** to show the list of files staged for commit.

```
> git status
```

```
On branch master
```

```
Your branch is ahead of 'origin/master' by 1 commit.
```

```
(use "git push" to publish your local commits)
```

```
Changes to be committed:
```

```
(use "git restore --staged <file>..." to unstage)
```

```
    modified:   git.tex
```

```
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:   git.pdf
```

```
    modified:   git.tex
```

The **git.tex** file has only some lines staged for commit.

Commits: what will be committed, but with more detail?

To see the details of the changes that are staged (that is, will be committed), you can run `git diff --cached`.

If you want to see the details of the changes that are *not* staged, you can run `git diff`.

`git diff` optionally accepts a list of files to diff.

```
diff --git a/git.tex b/git.tex
```

```
index 2c01a7b..91148d1 100644
```

```
--- a/git.tex
```

```
+++ b/git.tex
```

```
@@ -33,9 +33,7 @@
```

```
\section{Why use Git?}
```

```
-\begin{frame}{Why use Git? I}
```

```
-
```

```
- Example Scenario:
```

```
+\begin{frame}{Example Scenario 1}
```

```
\begin{enumerate}[<+>]
```

```
\item You start a project called ``my-proj'' and write a ton of code.
```

Commits: what will be committed, but with more detail?

To see the details of the changes that are staged (that is, will be committed), you can run `git diff --cached`.

If you want to see the details of the changes that are *not* staged, you can run `git diff`.

`git diff` optionally accepts a list of files to diff.

```
diff --git a/git.tex b/git.tex
```

```
index 2c01a7b..91148d1 100644
```

```
--- a/git.tex
```

```
+++ b/git.tex
```

```
@@ -33,9 +33,7 @@
```

```
    \section{Why use Git?}
```

```
-\begin{frame}{Why use Git? I}
```

```
-
```

```
-    Example Scenario:
```

```
+\begin{frame}{Example Scenario 1}
```

```
    \begin{enumerate}[<+>]
```

```
        \item You start a project called ``my-proj'' and write a ton of code.
```

Commits: showing existing commits

You can view a commit using `git show <commit hash>`.

```
commit cb1b9610e4d34aa66b52e7fb722654679b4529e2
```

```
Author: Sumner Evans <me@sumnerevans.com>
```

```
Date: Tue Jan 31 10:42:58 2023 -0700
```

```
matrix-synapse: 1.76.0rc2 -> 1.76.0
```

```
Signed-off-by: Sumner Evans <me@sumnerevans.com>
```

```
diff --git a/modules/services/matrix/synapse/default.nix
```

```
↪ b/modules/services/matrix/synapse/default.nix
```

```
index b142cf0..675ab77 100644
```

```
--- a/modules/services/matrix/synapse/default.nix
```

```
+++ b/modules/services/matrix/synapse/default.nix
```

```
@@ -7,20 +7,20 @@ let
```

```
    # Custom package that tracks with the latest release of Synapse.
```

```
    package = pkgs.matrix-synapse.overridePythonAttrs (old: rec {
```

```
      pname = "matrix-synapse";
```

```
-    version = "1.76.0rc2";
```

```
+    version = "1.76.0";
```

```
    format = "pyproject";
```

```
src = pkgs.fetchFromGitHub {
```

```
    owner = "matrix-org";
```

```
    repo = "synapse";
```

```
...
```

Use *git add* and *git reset* (or variants) to stage/unstage changes for your.

Use *git commit* to create a commit from the currently staged changes (which you can see by running *git diff --cached*).

Use *git add* and *git reset* (or variants) to stage/unstage changes for your.

Use *git commit* to create a commit from the currently staged changes (which you can see by running *git diff --cached*).

Branches



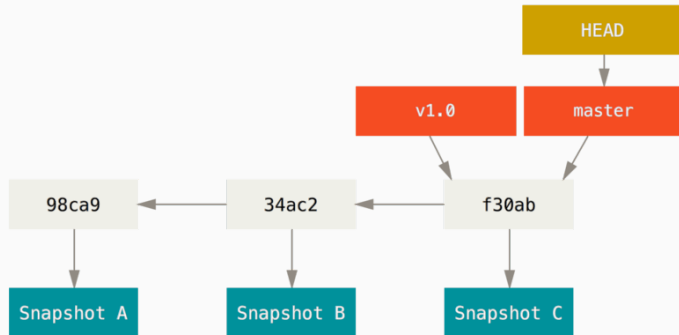
Branches: what are they?³

Remember how we said that the `.git` directory is like a “heap” for commits?

Branches are pointers to a specific commit in that heap. You can then follow that commit’s parent pointers to reconstruct the graph.

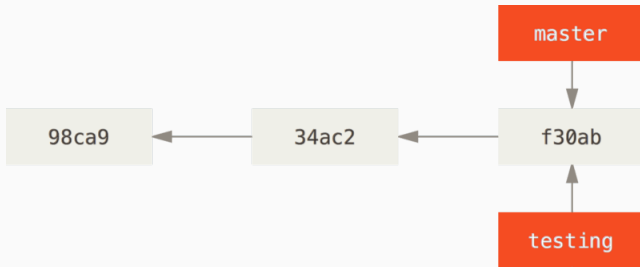
³Info in the rest of the *Branches* section is mainly from <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

Branches: pointers



Branches: creating them

Branches can be created using the `git branch <branch name>` command. This will not change your **HEAD** pointer.

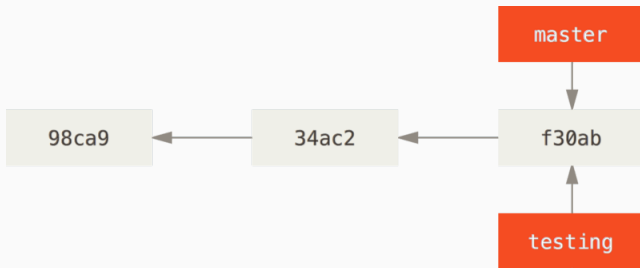


If you want to create a branch and also change the **HEAD** pointer to the newly created branch, you can use:
`git checkout -b <branch name>`.

You can use `git branch [-a]` to list (all) branches.

Branches: creating them

Branches can be created using the `git branch <branch name>` command. This will not change your **HEAD** pointer.



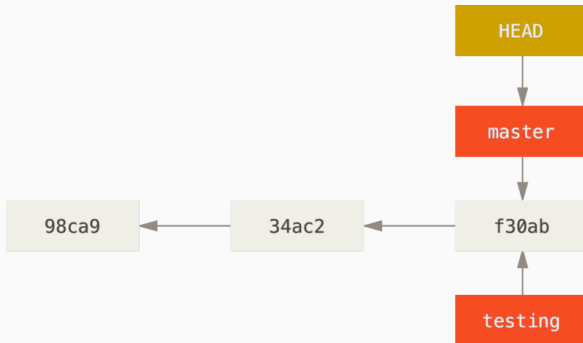
If you want to create a branch and also change the **HEAD** pointer to the newly created branch, you can use:
`git checkout -b <branch name>`.

You can use `git branch [-a]` to list (all) branches.

Branches: moving HEAD around

HEAD is a special pointer to the current repository state. Checking out a commit/branch will update the files in your working directory.

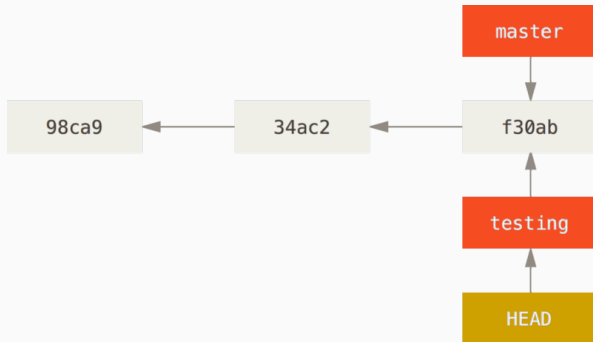
You can move the *HEAD* pointer to a different commit using *git checkout <commit hash or branch name>*.



Branches: moving HEAD around

HEAD is a special pointer to the current repository state. Checking out a commit/branch will update the files in your working directory.

You can move the **HEAD** pointer to a different commit using *git checkout <commit hash or branch name>*.



Branches: moving other branches around

If you want to move the branch that *HEAD* is pointing to to a different location, you can use *git reset*.

git reset --hard <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, discarding all changes in your working directory.

git reset --soft <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, leaving all changes since the specified commit as staged changes in your working directory.

Branches: moving other branches around

If you want to move the branch that *HEAD* is pointing to to a different location, you can use *git reset*.

git reset --hard <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **discarding all changes in your working directory**.

git reset --soft <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **leaving all changes since the specified commit as staged changes in your working directory**.

Branches: moving other branches around

If you want to move the branch that *HEAD* is pointing to to a different location, you can use *git reset*.

git reset --hard <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **discarding all changes in your working directory**.

git reset --soft <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **leaving all changes since the specified commit as staged changes in your working directory**.

Branches: checkout gotchas and pro-tips

- If you checkout a commit hash, you will be in a *detached HEAD* state because your *HEAD* pointer is not pointing to a branch.
- If you have uncommitted changes, switching branches *might* fail.

You can use *git stash* to save the changes in your working directory, then *checkout* the other branch, and then *git stash pop* to restore the changes.

Alternatively, you can just create a WIP commit and then switch to the other branch.

Pro tip: You can use `-` to refer to the previously checked out object.

Branches: checkout gotchas and pro-tips

- If you checkout a commit hash, you will be in a *detached HEAD* state because your *HEAD* pointer is not pointing to a branch.
- If you have uncommitted changes, switching branches *might* fail.

You can use *git stash* to save the changes in your working directory, then *checkout* the other branch, and then *git stash pop* to restore the changes.

Alternatively, you can just create a WIP commit and then switch to the other branch.

Pro tip: You can use *-* to refer to the previously checked out object.

Branches: checkout gotchas and pro-tips

- If you checkout a commit hash, you will be in a *detached HEAD* state because your *HEAD* pointer is not pointing to a branch.
- If you have uncommitted changes, switching branches *might* fail.

You can use *git stash* to save the changes in your working directory, then *checkout* the other branch, and then *git stash pop* to restore the changes.

Alternatively, you can just create a WIP commit and then switch to the other branch.

Pro tip: You can use `-` to refer to the previously checked out object.

Branches: checkout gotchas and pro-tips

- If you checkout a commit hash, you will be in a *detached HEAD* state because your *HEAD* pointer is not pointing to a branch.
- If you have uncommitted changes, switching branches *might* fail.

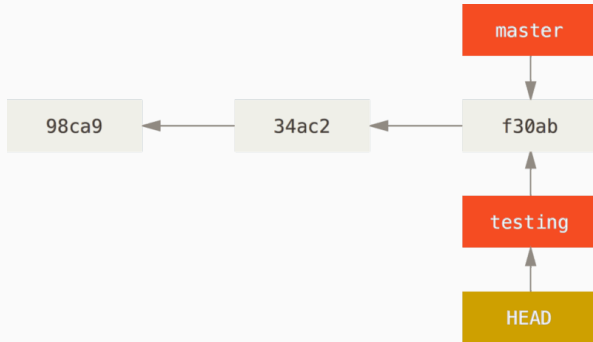
You can use *git stash* to save the changes in your working directory, then *checkout* the other branch, and then *git stash pop* to restore the changes.

Alternatively, you can just create a WIP commit and then switch to the other branch.

Pro tip: You can use `-` to refer to the previously checked out object.

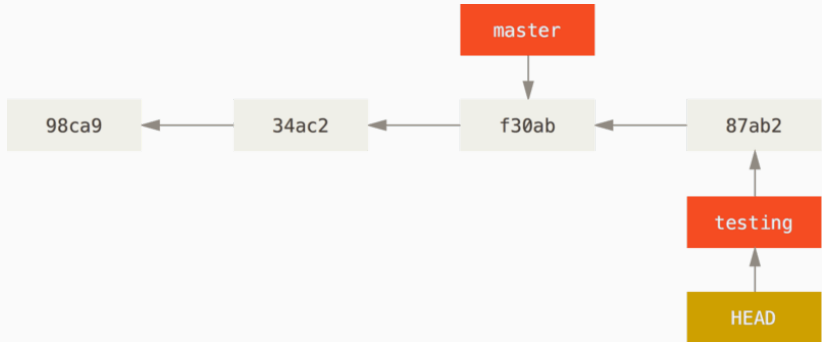
Branches: making commits

If you commit something while *HEAD* is pointed to a branch, both *HEAD* and your branch will move to the new commit.



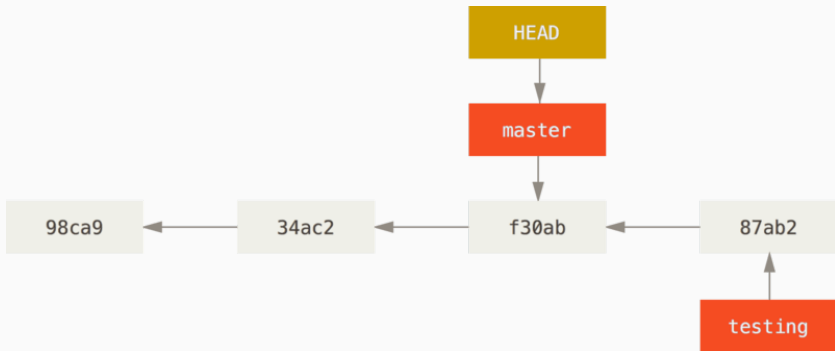
Branches: making commits

If you commit something while *HEAD* is pointed to a branch, both *HEAD* and your branch will move to the new commit.



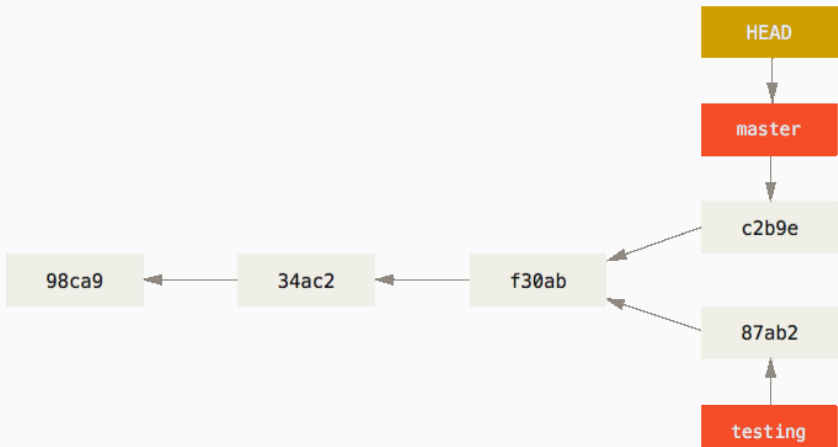
Branches: using multiple branches

Of course, you can always switch back to *master* using *git checkout master*.



Branches: divergence

If you make a commit on the master branch, the *master* pointer moves to that new commit creating **divergent** branch histories.



Branches: where am I?

Often, you want to get a summary of where you are in the repository. That's where *git log* comes in.

```
> git log
commit b08107c144003ba42495995d59234595d2d875b4 (HEAD -> master, origin/master,
↪ origin/HEAD)
Author: Sumner Evans <me@sumnerevans.com>
Date: Mon Feb 13 14:35:57 2023 -0700
```

fix some things

Signed-off-by: Sumner Evans <me@sumnerevans.com>

```
commit 6c1f8b53ac774dc6b0376810b4745951bd572519
Merge: 47bc626 67f0f4d
Author: Ethan Richards <42894274+ezrichards@users.noreply.github.com>
Date: Mon Feb 13 14:06:08 2023 -0700
```

Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com

...

This is mostly useless. Let's make it better.

Branches: where am I?

Often, you want to get a summary of where you are in the repository. That's where *git log* comes in.

```
> git log
commit b08107c144003ba42495995d59234595d2d875b4 (HEAD -> master, origin/master,
↪ origin/HEAD)
Author: Sumner Evans <me@sumnerevans.com>
Date: Mon Feb 13 14:35:57 2023 -0700
```

fix some things

Signed-off-by: Sumner Evans <me@sumnerevans.com>

```
commit 6c1f8b53ac774dc6b0376810b4745951bd572519
Merge: 47bc626 67f0f4d
Author: Ethan Richards <42894274+ezrichards@users.noreply.github.com>
Date: Mon Feb 13 14:06:08 2023 -0700
```

Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com

...

This is mostly useless. Let's make it better.

git log: but actually good

For `git log` to be useful, you want it to show *all* branches, show a graph, and get rid of most of the details.

```
> git log --all --graph --decorate --oneline
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
*   6c1f8b5 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 67f0f4d fix background color bug
* | 47bc626 Fix accordion
|/
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
* 3d0f64a Archive preliminary updates
* aaaa02b Update FAQ and archive pages
*   b2a64f7 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 1450745 make footer reveal
* | f7ddfa2 Fix alt text
|/
* 0f89721 created student confirm registration page
* 7b15e86 editing teams: ensure that you can't change from in-person to remote or
↪ vice versa
* e15bda6 save team below member list
* 853dcd1 add ability to add team members
```

Branches: summary

- Branches are **pointers** to commits.
- Use `git checkout` to move between branches.
- Use `git log` to see where you are.

Merging



Merging: resolving divergent histories⁴

If you want to merge the changes from branch *A* into another branch *B*, you need to:

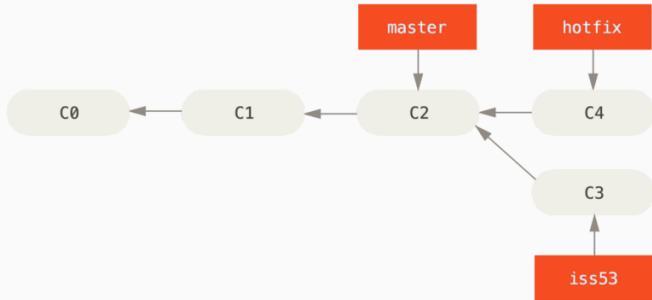
1. Switch to branch *B* (`git checkout B`)
2. Run `git merge A`.

This will do one of two things: fast-forward or create a merge commit.

⁴Info in the rest of the *Merging* section is mainly from <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

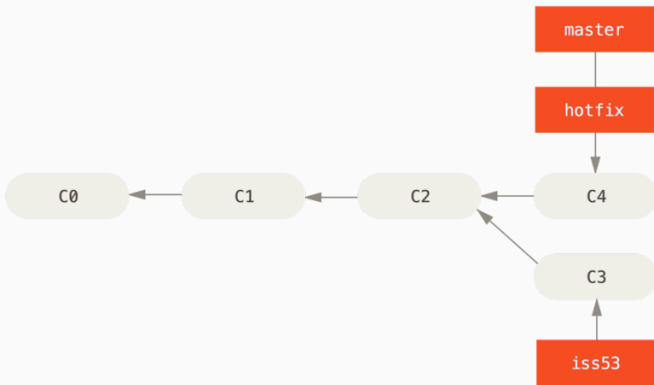
Merging: fast-forwarding

If the branch you are merging is directly ahead of the branch you are merging into, Git will just move the pointer in a **fast-forward merge**.



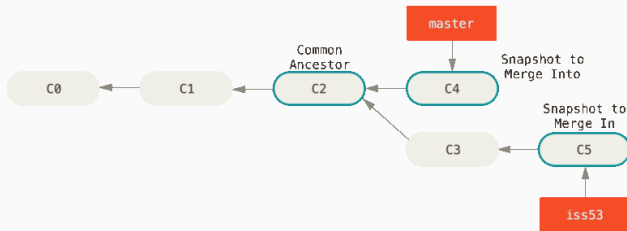
Merging: fast-forwarding

If the branch you are merging is directly ahead of the branch you are merging into, Git will just move the pointer in a **fast-forward merge**.



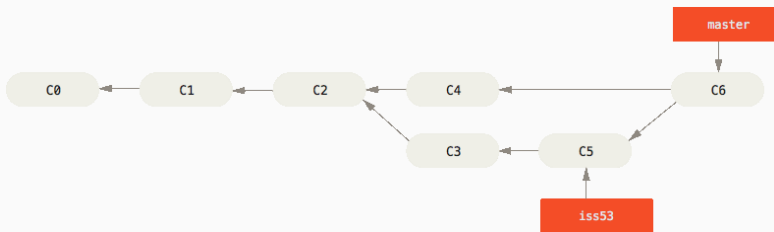
Merging: creating a merge commit

If the branch you are merging has diverged from the one you are merging into, Git will create a merge commit through a **three-way merge**.



Merging: creating a merge commit

If the branch you are merging has diverged from the one you are merging into, Git will create a merge commit through a **three-way merge**.



Merging: resolving conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly.

```
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

You can always use *git status* to see what has been automatically merged and what files have conflicts.

```
> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

    both modified:   index.html

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

Merging: resolving conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly.

```
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

You can always use *git status* to see what has been automatically merged and what files have conflicts.

```
> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

    both modified:   index.html

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

Merging: editing files to resolve merge conflicts

You can use a merge tool to resolve conflicts, however I find that it's easier to just manually resolve the conflicts.

Visual Studio Code has a good UI for this. The process you should follow is as follows:

1. Open a file with the conflict.
2. Find one of the conflict-resolution markers.
3. Make edits to resolve the conflict.
4. Run `git add` on the file.
5. Repeat steps 1-4 until all conflicts are resolved.
6. Run `git commit` to commit the merge.

Merging: understanding conflict-resolution markers

In order to find the conflict-resolution markers, search for <<<<<<. Each conflict-resolution block should look something like this:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

The first part (between <<<<<< and =====) is what the branch you are merging *into* has. The second part (between ===== and >>>>>>) is what the branch you are merging *from* has.

Sometimes, you just one one side of the conflict, other times you need to be more nuanced in your merge.

Merging: understanding conflict-resolution markers

In order to find the conflict-resolution markers, search for <<<<<<. Each conflict-resolution block should look something like this:


```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

The first part (between <<<<<< and =====) is what the branch you are merging *into* has. The second part (between ===== and >>>>>>) is what the branch you are merging *from* has.

Sometimes, you just one one side of the conflict, other times you need to be more nuanced in your merge.

- Merging allows you to pull changes from one branch into another branch.
- Use `git merge A` to merge branch *A* into the current branch.
- Git will do a fast-forward merge if possible, otherwise it will create a merge commit.
- You might have to resolve merge conflicts.

Rebasing



Rebasing: maintaining linear histories

Most merge commits are useless. They clutter the history, and normally don't add anything of value to the understanding of how the codebase evolved.

```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
* 6c1f8b5 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 67f0f4d fix background color bug
* | 47bc626 Fix accordion
|/
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

Ideally the above history would be linear:

```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
* 47bc626 Fix accordion
* 67f0f4d fix background color bug
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

Rebasing: maintaining linear histories

Most merge commits are useless. They clutter the history, and normally don't add anything of value to the understanding of how the codebase evolved.

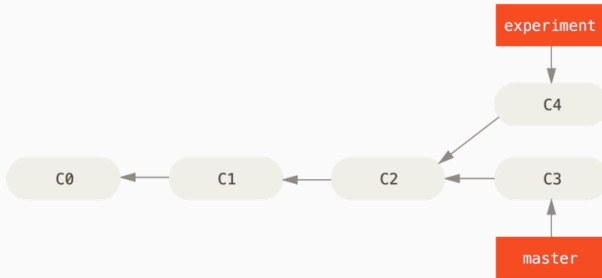
```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
* 6c1f8b5 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 67f0f4d fix background color bug
* | 47bc626 Fix accordion
|/
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

Ideally the above history would be linear:

```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
* 47bc626 Fix accordion
* 67f0f4d fix background color bug
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

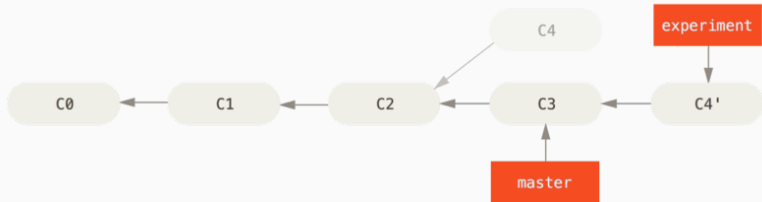
Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



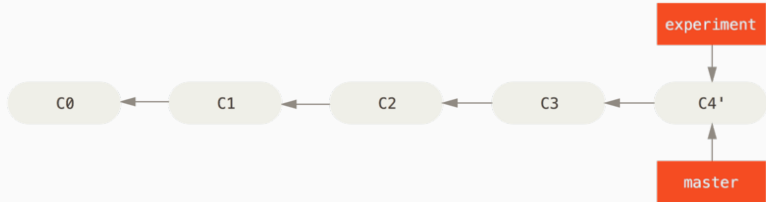
Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



To rebase, follow this procedure:

1. Checkout the branch that you want to rebase.
2. Run `git rebase A`, where *A* is the branch or commit you want to rebase onto.

Note that **rebasing rewrites history**. Generally you should only rebase *local* commits, or branches that you are the only one using.

To rebase, follow this procedure:

1. Checkout the branch that you want to rebase.
2. Run `git rebase A`, where *A* is the branch or commit you want to rebase onto.

Note that **rebasing rewrites history**. Generally you should only rebase *local* commits, or branches that you are the only one using.

Rebasing: interactive rebase

Say you are working on a feature branch and you've made the following commits:

```
* 1a2b3c4 (HEAD -> my-feature) add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

But you realize that your *fix fizzing* commit introduced a bug! So you fix the issue but you don't want to end up with a "fix the fix" commit.

Interactive rebasing can help! Go ahead and create a new "fix the fix" commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Rebasing: interactive rebase

Say you are working on a feature branch and you've made the following commits:

```
* 1a2b3c4 (HEAD -> my-feature) add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

But you realize that your *fix fizzing* commit introduced a bug! So you fix the issue but you don't want to end up with a "fix the fix" commit.

Interactive rebasing can help! Go ahead and create a new "fix the fix" commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Rebasing: interactive rebase

Say you are working on a feature branch and you've made the following commits:

```
* 1a2b3c4 (HEAD -> my-feature) add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

But you realize that your *fix fizzing* commit introduced a bug! So you fix the issue but you don't want to end up with a "fix the fix" commit.

Interactive rebasing can help! Go ahead and create a new "fix the fix" commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Rebasing: interactive rebase (continued)

Interactive rebasing can help! Go ahead and create a new “fix the fix” commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Now, run *git rebase -i master* to start an interactive rebase. This will open an editor with the following (as well as instructions):

```
pick 8a9b0c1 fix fizzing
pick 1a2b3c4 add more buzz
pick a81abe1 fixup! fix fizzing
```

Now, you can move the commit order around by editing the file, and also *fixup* the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

Rebasing: interactive rebase (continued)

Interactive rebasing can help! Go ahead and create a new “fix the fix” commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Now, run ***git rebase -i master*** to start an interactive rebase. This will open an editor with the following (as well as instructions):

```
pick 8a9b0c1 fix fizzing
pick 1a2b3c4 add more buzz
pick a81abe1 fixup! fix fizzing
```

Now, you can move the commit order around by editing the file, and also ***fixup*** the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```


Rebasing: interactive rebase result

Now, you can move the commit order around by editing the file, and also ***fixup*** the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

After saving and exiting the editor, Git will rewrite the history of your branch to look like this:

```
* a9bb207 (HEAD -> my-feature) add more buzz
* 4e89d4f fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

and the ***fix fizzing*** commit will contain the changes from both commits!

Note that you now have different commit hashes! The old commits are still in tact (you can ***checkout*** them), but you have moved your branch (pointer) to the new commit.

Rebasing: interactive rebase result

Now, you can move the commit order around by editing the file, and also **fixup** the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

After saving and exiting the editor, Git will rewrite the history of your branch to look like this:

```
* a9bb207 (HEAD -> my-feature) add more buzz
* 4e89d4f fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

and the **fix fizzing** commit will contain the changes from both commits!

Note that you now have different commit hashes! The old commits are still in tact (you can *checkout* them), but you have moved your branch (pointer) to the new commit.

Rebasing: interactive rebase result

Now, you can move the commit order around by editing the file, and also **fixup** the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

After saving and exiting the editor, Git will rewrite the history of your branch to look like this:

```
* a9bb207 (HEAD -> my-feature) add more buzz
* 4e89d4f fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

and the **fix fizzing** commit will contain the changes from both commits!

Note that you now have different commit hashes! The old commits are still in tact (you can **checkout** them), but you have moved your branch (pointer) to the new commit.

Rebasing: interactive rebase shortcut

The workflow I described in the previous slides is so common that there are tools built-in to Git to help you accomplish them.

- `git commit --fixup <commit>` automatically marks your commit as a fix of a previous commit. (It uses that *fixup!* syntax from the previous slide.)
- `git rebase -i --autosquash` opens the editor and automatically reorders the fixup commits when the interactive rebase editor opens and sets them to *fixup* instead of *pick*.

See <https://fle.github.io/git-tip-keep-your-branch-clean-with-fixup-and-autosquash.html> for more details.

If you ever need to cancel a rebase, use `git rebase --abort`.

This will restore your repository state to what it was before you started the rebase process.

There are lots of other reasons to rebase.

- You want to pull in changes from another branch without merging.
- You want to modify a commit that added a file by accident.
- You want to reorder your commits to make it more clear to a code reviewer what you changed.
- Somebody else pushed a commit to the *remote* branch, and you want to add your commit on top of theirs.

Remotes



Remotes: what are they?⁵

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

GitHub is a popular choice for where to host your remote repositories, but other options exist such as GitLab, sourcehut, Bitbucket, and many other self-hosted options.

⁵Much of the content of this section is from <https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes>

Remotes: what are they?⁵

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

GitHub is a popular choice for where to host your remote repositories, but other options exist such as GitLab, sourcehut, Bitbucket, and many other self-hosted options.

⁵Much of the content of this section is from <https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes>

Remotes: managing them

- View all of your remotes with:

```
> git remote -v
```

```
origin  git@github.com:sumnerevans/acm-git-good.git
```

```
↪ (fetch)
```

```
origin  git@github.com:sumnerevans/acm-git-good.git
```

```
↪ (push)
```

- You can add remotes using:

```
> git remote add other git@github.com:other/repo.git
```

- You can change the URL of a remote:

```
> git remote set-url origin
```

```
↪ git@github.com:other/repo.git
```

- When you clone a repository, it automatically creates a remote called *origin* that points to your remote repository.
- *Note:* *origin* is not special, it's just a default.

Remotes: managing them

- View all of your remotes with:
 `> git remote -v`
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (fetch)
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (push)
- You can add remotes using:
 `> git remote add other git@github.com:other/repo.git`
- You can change the URL of a remote:
 `> git remote set-url origin`
 ↪ `git@github.com:other/repo.git`
- When you clone a repository, it automatically creates a remote called *origin* that points to your remote repository.
- *Note: origin* is not special, it's just a default.

Remotes: managing them

- View all of your remotes with:
 `> git remote -v`
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (fetch)
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (push)
- You can add remotes using:
 `> git remote add other git@github.com:other/repo.git`
- You can change the URL of a remote:
 `> git remote set-url origin`
 ↪ `git@github.com:other/repo.git`
- When you clone a repository, it automatically creates a remote called *origin* that points to your remote repository.
- **Note:** *origin* is not special, it's just a default.

Remotes: managing them

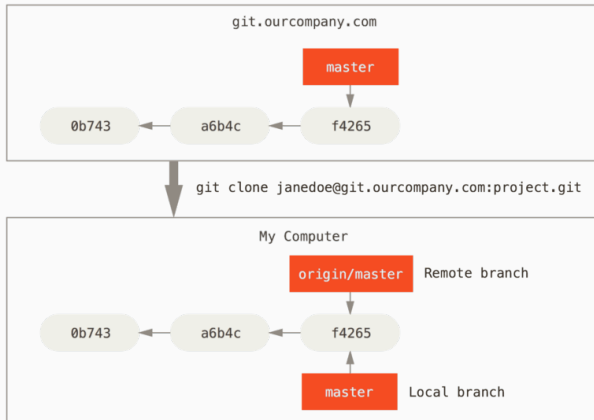
- View all of your remotes with:
 `> git remote -v`
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (fetch)
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (push)
- You can add remotes using:
 `> git remote add other git@github.com:other/repo.git`
- You can change the URL of a remote:
 `> git remote set-url origin`
 ↪ `git@github.com:other/repo.git`
- When you clone a repository, it automatically creates a remote called **origin** that points to your remote repository.
- *Note: **origin** is not special, it's just a default.*

Remotes: managing them

- View all of your remotes with:
 `> git remote -v`
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (fetch)
 `origin git@github.com:sumnerevans/acm-git-good.git`
 ↪ (push)
- You can add remotes using:
 `> git remote add other git@github.com:other/repo.git`
- You can change the URL of a remote:
 `> git remote set-url origin`
 ↪ `git@github.com:other/repo.git`
- When you clone a repository, it automatically creates a remote called **origin** that points to your remote repository.
- **Note:** **origin** is not special, it's just a default.

Remote Branches: what are they?

The remote repository is **entirely separate** from your **local repository**. The remote repository has its own set of branches, commits, etc.



1. **Fetch:** to download the current state of the remote repository, use the *git fetch* command.
Git does not automatically fetch the state of the remote repository!
2. **Push:** to update the remote repository with your local state, use the *git push* command.
3. **Tracking branches:** tracking branches are local branches that have a direct relationship to a remote branch.

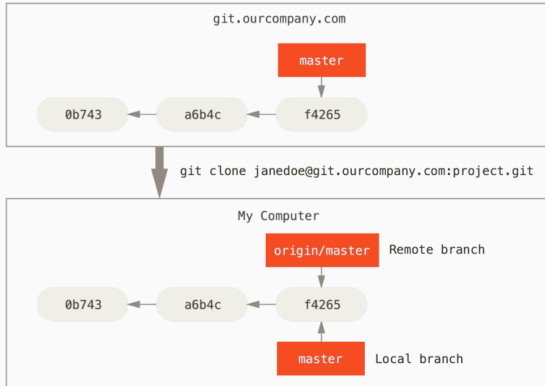
1. **Fetch:** to download the current state of the remote repository, use the *git fetch* command.
Git does not automatically fetch the state of the remote repository!
2. **Push:** to update the remote repository with your local state, use the *git push* command.
3. **Tracking branches:** tracking branches are local branches that have a direct relationship to a remote branch.

Remote Branches: three new concepts

1. **Fetch:** to download the current state of the remote repository, use the *git fetch* command.
Git does not automatically fetch the state of the remote repository!
2. **Push:** to update the remote repository with your local state, use the *git push* command.
3. **Tracking branches:** tracking branches are local branches that have a direct relationship to a remote branch.

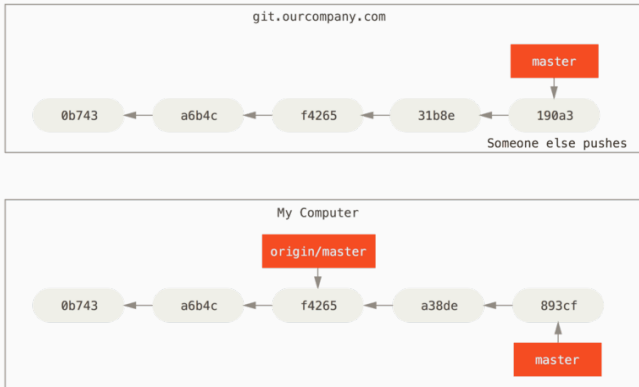
Remote Branches: divergence

The state of the local and remote repositories can diverge.



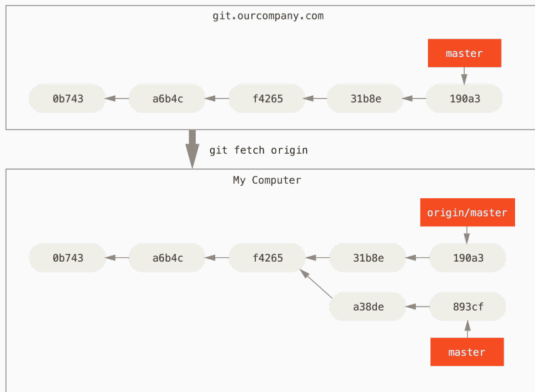
Remote Branches: divergence

The state of the local and remote repositories can diverge.



Remote Branches: divergence

The state of the local and remote repositories can diverge. Use *git fetch* to get the latest state of the remote repository.



Now use the tools we already know for divergent branches.

Remote Branches: divergence (continued)

Git provides a command to fetch and then merge the remote branch into the local branch called *git pull*.

Warning: by default, *git pull* will create a merge commit if the remote branch has diverged from the local one!

Merge commits are ugly! Use *git pull --rebase* to tell *git pull* to rebase your local changes on the remote changes instead.

Tell Git to never merge when pulling

You can configure Git to fail instead of making a merge commit by setting the *pull.ff* configuration option to *only*.

```
> git config --global pull.ff only
```

Remote Branches: divergence (continued)

Git provides a command to fetch and then merge the remote branch into the local branch called *git pull*.

Warning: by default, *git pull* will create a merge commit if the remote branch has diverged from the local one!

Merge commits are ugly! Use *git pull --rebase* to tell *git pull* to rebase your local changes on the remote changes instead.

Tell Git to never merge when pulling

You can configure Git to fail instead of making a merge commit by setting the *pull.ff* configuration option to *only*.

```
> git config --global pull.ff only
```

Remote Branches: divergence (continued)

Git provides a command to fetch and then merge the remote branch into the local branch called *git pull*.

Warning: by default, *git pull* will create a merge commit if the remote branch has diverged from the local one!

Merge commits are ugly! Use *git pull --rebase* to tell *git pull* to rebase your local changes on the remote changes instead.

Tell Git to never merge when pulling

You can configure Git to fail instead of making a merge commit by setting the *pull.ff* configuration option to *only*.

```
> git config --global pull.ff only
```


Remote Branches: pushing

Pushing can be thought of merging your local tracking into the remote tracking branch.

If your branch is already configured to track a remote branch, you can just use *git push*.

If you have a new branch that doesn't have a corresponding remote tracking branch, use

```
> git push -u origin <branch name>
```

Pushing can be thought of merging your local tracking into the remote tracking branch.

If your branch is already configured to track a remote branch, you can just use *git push*.

If you have a new branch that doesn't have a corresponding remote tracking branch, use

```
> git push -u origin <branch name>
```

Remote Branches: pushing harder

In certain circumstances, you want to push a divergent branch to the remote. Examples include:

- You did an interactive rebase and need to push the newly rebased changes.
- You reset your branch to a previous commit and want it to be the latest commit on the branch.

In these cases, you can use the `--force` or `--force-with-lease` options to tell Git to push your local state to the remote regardless of what is currently there. `--force-with-lease` is less dangerous than `--force` because it will check that your current view of the remote state is up-to-date.

Remote Branches: pushing harder

In certain circumstances, you want to push a divergent branch to the remote. Examples include:

- You did an interactive rebase and need to push the newly rebased changes.
- You reset your branch to a previous commit and want it to be the latest commit on the branch.

In these cases, you can use the `--force` or `--force-with-lease` options to tell Git to push your local state to the remote regardless of what is currently there.

`--force-with-lease` is less dangerous than `--force` because it will check that your current view of the remote state is up-to-date.

Advanced Tips

Commits are kept around in the `.git` directory (remember, it's like a heap), which means that even if you lose a pointer to a commit (for example by moving all branch pointers away from it), it still exists!

You can use `git reflog` to see the history of when the tips of branches and other references were updated in the local repository.

If you screwed something up and you don't know what to do, <https://ohshitgit.com/> is a great resource.

Undoing Things

Commits are kept around in the `.git` directory (remember, it's like a heap), which means that even if you lose a pointer to a commit (for example by moving all branch pointers away from it), it still exists!

You can use `git reflog` to see the history of when the tips of branches and other references were updated in the local repository.

If you screwed something up and you don't know what to do, <https://ohshitgit.com/> is a great resource.

Cherry-picking

You can apply arbitrary commits to your current branch by cherry-picking them.

This is similar to rebasing.

Git supports the concept of *aliasing* one command to another name.

For example, you can alias the pretty log command I showed earlier to ***git l*** by using the following command:

```
> git config --global alias.l "log --oneline --graph --all  
↪ --decorate"
```

Now you can just type ***git l*** to get the pretty version of the log output.

You can tell Git to not ever commit certain files by ignoring them using the *.gitignore* file.

Any file that matches one of the patterns in *.gitignore* will not be tracked by Git.

Warning: if you have already committed the file, it will still be tracked.

If you want to delete a file from Git, but keep it on your computer, use *git rm --cached <file>*.

You can tell Git to not ever commit certain files by ignoring them using the `.gitignore` file.

Any file that matches one of the patterns in `.gitignore` will not be tracked by Git.

Warning: if you have already committed the file, it will still be tracked.

If you want to delete a file from Git, but keep it on your computer, use `git rm --cached <file>`.

Want to see the last commit that modified each line of a file?

Use *git blame*.

This is often a helpful way to see which commit cause a bug or see what needs to be changed when making a similar change.

I obviously was unable to tell you about everything you can do with Git. There are hundreds of options on every single command.

- `man git *`: The man pages on Git are good. Use them as your first line of defense.
- git-scm.com/book/en/v2: A huge resource about how to do everything Git.
- gitignore.io: Generates a `.gitignore` file for a given project type, OS, and IDE.
- `delta`: provides a much prettier *diff* interface.