**Github Mastery**

**Command index:**

* **git checkout**
  + moves header pointer from one commit to another
  + moves data from repository into working area based on the commit you checked out
* **git add**
  + simply moves data from the working area to the index
* **git status**
  + see if working area is up to date with the index area
* **git diff**
  + more-detailed comparison of working area to index area
* **git diff --cached**
  + shows difference between the index (things that you added) and the repository (things that you’ve committed). Is more-detailed than git status
* **branch**
  + list branches or commits that you can move your header-pointer to with checkout
* **git rm**
  + removes file from index AND working area (undoes a ‘git add’ AND tries to remove the file)
    - there are safety features to prevent you from git rm’ing files that are not in the respository yet
* **git rm –cached**
  + removes a file from the staging area (index) but NOT the working area. Basically JUST undoes a ‘git add’
* **git mv**
  + tries to rename/move a file in the working area AND the index
    - it’s better to just ‘git add’ both the new file and the non-existant old file (yes you can do this) to sync up the index and working area
* **git reset –hard** 
  + will copy data from the new current-commit into the working area and index area.
    - Git reset --hard <commit hash>
      * this will set your working area and index BACK IN TIME to a commit that you had. The commits SINCE that commit will eventually be deleted as garbage
* **git Reset –mixed** 
  + will copy data from the new current commit to the index BUT won’t touchyour current working area. This is the default option
* **git reset –soft** 
  + doesn’t touch the other areas. It just moves the branch and doesn’t touch the index or working area.
* **git reset HEAD**
  + copiesdata from the current repository (where HEADER is pointing) into the staging area (index) but NOT the working area
* **git reset –hard HEAD**
  + same as above only this will update the working area as well
    - very popular command to say “just undo everything and set me up as the last commit was set for”
* **git log**
  + print commits and their hashes
* **git stash –include-untracked**
  + this will store your current state so that you can work on other things!
  + copies the working area AND the index and stashes it!
    - THEN it checks out the current commit again!
* **Git stash –include-unstracked** 
  + will stash your current state and set you back to the clean state of the current commit!
* **Git stash list** 
  + lists all of your stashes
  + So the logical question now is “how do I restore the state from my git stash?”
    - our half-done work stays in the stash and we go off to do some other work
* **git stash apply <label from list>**
  + restores working area and index to the state from the stash that was stored when you stashed it
  + and there we are! All of our data is back in the working area and index as it was when we stashed it for us to continue with! Very powerful! No more need to have multiple git directories!!!
* **git reset HEAD menu.txt**
  + will make menu.txt be whatever it was in the repo IN THE STAGING AREA but not the working area
  + see below to find out how to make changes to index AND working areas from the repository, you cannot use git reset with –hard for a path!!
* g**it checkout HEAD menu.txt** 
  + checkout will not move head, just pull the current info from the repository into the working area and index!!!
* **Git log –graph**
  + will display git log in a new graph form
  + **git log –decorate** will make it look better
    - **git log –decorate –oneline** will condense it
* **git show <hash>** 
  + will show you a much more technical explanation of what a commit changed!
* g**it show <branch>** also does this.
* **git show HEAD^**
  + will show you the parent commit to head! You can climb this way.
* **git show HEAD ^^**
  + will show you the parent of the parent! This is fine for as far back until you have commits with multiple parents/comits before them!
* **git show HEAD^2^2**
  + will do the second parent of TWO commits back
* **git show HEAD(9’1 month ago’)**
  + will show you the commit from one month ago! Very cool!
* **git blame <filepath>** 
  + will show you the changed lines and the LATEST commit that changed those files (very cool)!
* **git diff HEAD HEAD^2** 
  + will show you the change of the current commit vs two commits earlier!
* **git diff <branch> <branch2>** 
  + will diff two branches before commits!
* **git log –patch**
  + will give you a DETAILED git log that’s color-coded.
* g**it log –grep apples –oneline** 
  + will only show log info for lines containing “apples”
* g**it log -Gapples –patch** 
  + will show lines impacted by changes that either added or deleted ‘apples’ in the line.
  + **git log -3 –online**
    - limits range to last three commits
* **git rebase -i <branch to start with>**
  + will bring you into a list of commits in the opposite of local order. Do not get confused. Now time to interractively fix it. We can force our commit into the one that was wrong in the past!
  + you’ll see:
    - pick <hash> <description>
    - pick <hash> <description>
    - etc.. etc……
      * and you’ll also see a list of commands you can replace “pick” with and what they’ll do. Many people do many commits, rebase, fix descriptions or squash (combine) commits, THEN do the push to origin!
      * Tested and working, this is very cool/powerful
* **git reflog HEAD**
  + will tell you all of the changes that happen to HEAD (pointer to current branch, remember).
  + This is like ‘history’ but for git only. Very Cool! But it’s not meaningful! Look at the numbers though! We can use this with **git show**
  + **git reflog *refs/*heads/master** 
    - will show you when the local master branch was created. Hopefully that will make you feel safer because now you are able to recover and find your old hashes thanks to the reflog!
    - To recover these just checkout the hash you discover from using reflog and commit it back into the mix!
* **git revert <bad commit’s hash id>**
  + creates a brand new commit that is just changes that are the opposite of all of the changes from the commit whose hash you gave. It’s absolutely amazing. There is a new commit. The badness stays in history, but it’s better than editing a shared commit!

**Mastering Git intro:**

We will use sophisticated commands and at the end hopefully be a git master. This course will allow you to know github at a level where you can lead your team and become a power-user. This is an advanced git course, not for the basics. It also focuses the “git way of thinking”.

**The four areas of git:**

There is no point in memorizing the tons of git commands as there is always ‘git help’ - we will focus on “the way that git thinks” and what really happens when you do these common operations.

If git was an onion this lesson would be about cooking the onion. We don’t talk about how git stores your data, we talk about how git moves your data and manages it.

You move data across four areas:

1. introduction

2. Basic workflow

3. git reset (all about it) one of the most powerful and confusing commands in git.

4. advanced git tools

5. exploring the history of a project

6. fixing mistakes

7. finding YOUR workflow for YOUR team (fluffy he says)

The entire course will be used through the command-line. Believe it or not, it’s just easier this way to become a git master.

**The Four Areas:**

1. Working area

where you keep your current files and folders

2. Repository

‘main area where you store entire history of the project’

‘most important area’

3. Index

‘where you put your files before your commit’

essential to understand how to use this

4. Stash

‘temporary storage area’

useful, but not nearly as important as the others

Two important questions for every single git command.

*How does this command move information across the four areas?*

*How does this command change the Repository?*

It doesn’t matter how scary a command looks. If you can answer these two questions you’ll be able to master git.

**Quick review of the working area:**

The project on the filesystem. You work here, test here, etc…

Let’s make and edit some random file. Git doesn’t care too much about the working area at any time. Git respects it, avoids destroying it, but in this training we’ll see a few commands that will destroy the working area. Don’t assume data is safe until it’s committed. Once you commit, it’s stored into the important area. The repository.

**The repository:**

**ls -la to see the .git directory**

the most important part of this sub-directory is the ‘objects’ database in the objects folder. There are a few different kinds of objects in here.

* Trees
  + represent folders
* blob
  + represents files
* commits
  + represents… commits

These commit points to trees which point to blobs. Each commit is a snapshot of your working area at a certain point in time. Two commits can share the same object if the object has not changed!! That’s the way that git stores changes to your files and directories. Keep this in mind.

Each commit is also pointing to its parent commit to create a history. Everything eventually pointing down to the first commit in the chain. So if each commit is a snapshot, and the commit history is a history of your project history through snapshots, that’s awesome.

A BRANCH is a reference to a commit. A branch is the entry-point to a history of commits! You can have multiple slices of history. The same commits can belong to multiple branches this way.

There is a special pointer called “HEAD”. There can be only one “HEAD”. It’s usually pointing to a commit which is the ‘current’ branch. If you move ‘head’ (checking out another branch) then you move current-commits.

Sometimes you do operations to make commits not-reachable. Git eventually deletes this garbage.

That was a very quick recap of the working area and the repository.

**The Index:**

The index is also called the “Staging area”. You don’t go right from the working area to the repository, you stage it first in the index area!

If you run ‘git status’ you’ll see that there’s nothing new in the working area. We’ll call this the “clean status”. The working area, index, and repo are all aligned!

These files are in the current commit of the repository. The folders are trees and the files are blobs. Keep this in mind.

If you look in the subgit directory you’ll see its a binary file you can’t see. So you have to visualize.

Right now the index is seen as a staging area right? You add them into the index, you launch them into the repository by committing. You need to think of it differently.

Index is ‘just another area that holds everything’.

A clean state in ‘git status’ doesn’t mean that the index is empty, it just means that the index contains the same files and folders as the repository. It’s a small mental shift BUT it’s important. The three areas are currently all alligned.

**‘git diff’ -** shows difference between stages. Instead we should compare the index with the repository.

‘**git diff –cached’** - this option shows the difference between the index (things you’ve changed) and the repository (things you’ve committed)

Trying out ‘git diff –cached’ is amazing (two dashes there btw). It shows the differences between things i’ve staged for commit vs the things that I’ve actually committed. It’s like a middleman from ‘git status’

**The Basic Workflow:**

Remember the two golden questions?

1. How does this command move information across the four areas?

2. How does this command change the repository?

Let’s look at these commands through the scope of these two questions.

**Moving Data ‘to the right’:**

You already know the basic workflow of git. Edit file, stage file, commit file.

Let’s make an edit to a text file in our branch.

Git status will tell us that the file has been staged but it’s not in the index yet! It’s been modified. Let’s add it. Check git diff as well.

Git add <file> will do this. You know that. Now git status will show that it’s staged and in the index and the index looks in sync with the working area.

Git commt – let’s commit it, copying it from the index to the respository. We’ve y.moved the file from the working area, to the index, to the repository. As a result, git diff will show no difference nor will git diff –cached. We’re in the clean state again.

Git commit also updates the current branch and creates a new commit altogether. It moves data and also changes the repository. By contrast, ‘git add’ just moves data and doesn’t touch the repository

**Moving Data ‘to the left’:**

How do we move data backwards? There is a common one! CHECKOUT!

Git checkout moves the header-reference to another branch

Git checkout also takes data from the new current commit and copies the data from the repo to the index to the working area!

Run ‘git branch’ to view our branches! Let’s look at some. Make sure to commit first so there’s some branches here to look at! And be aware of where your header is.

‘Head’ is pointing to the ‘master’ branch when we’re here.

Let’s check out a new branch.

The head moves and the repository has been altered by the checkout! Now the repo is the new branch but index and working area has changed! (the repo did NOT change, just we’re looking at different trees and blobs)

Now the index and working area were updated as well.

Back to the master branch.

We already knew ‘checkout’ - but now we have a better understanding for what it does. Do not confuse ‘git’ with ‘github’! This is not something you’ve done many times at work.

**Removing Files:**

Let’s make a new file and fill it with some text. I plan to commit it, so let’s git add it so it’s over in the index now! Git status confirms. It’s in the working area and index, but not in the repository yet.

Now, what if we change our mind. What if we want to remove the file from the index BUT remove it from the index??? Maybe we want to commit something FIRST but keep this file in the working area? How can we accomplish this?

g**it rm –** is actually NOT the answer…. Git rm will try to delete the file from the working area AND the index. It’s destructive, but has safety features if the file is not in project-history.

→ you can force this removal with –f orrr can remove it from the index alone with –cached

**git rm –cached** is what we want to use! This will remove the file from the staging area (the index) but NOT the working area!!! You ‘unstage’ the file.

To recap, git rm is not the opposite of git add. Git rm –cached (two dashes) is the true opposite of ‘add’. Seems confusing, but trust it for now.

After you’ve successfully removed your file from the index, you can just use regular **rm** to get rid of the file from the working area and git status will show you have returned to a clean state.

**Moving and Renaming Files:**

Like linux, renaming a file is just ‘moving it in-place’. It’s the same here.

Let’s say we want to change the name of one of our files that’s been there for a while? Let’s rename it normally in the working area with **mv**

Now let’s try **git status –** it’s confused? It sees a deleted file and a new file. This isn’t quite true! How can we explain to git that these two files are the same file with the same name?

First let’s move the new file into the index with **git add**

Now (hold onto your hats) **git add the old file name –** yes seriously. We’re git add’ing something no longer in the working area. So if the working area doesn’t have it, it overrides the data in the index with nothing! Now we have the same data in the working area and the index. **Git status** now understands automatically that this was a rename after doing this! The ‘new’ file is now a ‘renamed’ file! That’s pretty smart. It works for renaming AND moving.

You can also use this to change AND rename files at the same time. Unless you completely change a file ENTIRELY, git can tell when you’re moving/renaming a file that has a few edits as well. It just works.

SO. **git add the renamed/moved file** then **git add the old file that no longer exists**

**git mv** attempts to wrap all of this up but the instructor notes that it’s better to use the multiple git add’s as it’s better to follow through with what’s happening this way.

**A Quick Summary**:

‘add’ copies data from working area to index

‘commit’ copies data from the index into the repository and creates new objects in the repo

‘checkout’ copies data from the repository into the indexd and the working area

‘remove’ takes data out of working area AND index

‘mv’ tries moving data in the working area AND the index

**NEW MODULE: Understanding ‘Reset’:**

Git reset is one of the most-powerful tools that git has. Understanding it will make you a power-user. Make sure to understand what it does.

**Reset** is the main tool that explicitly just moves a branch.

To understand it you need to understand the way that branches work and the four stages of git. The good news is that we just reviewed it above so we’re ready to go.

Reset also has many many use-cases, so examples might not be so relatable at first. You end up thinking that it must be a very complex command, but this isn’t true.

**Commands that move branches, and why reset is different:**

* commit
  + creates new commit and moves branch to point at the new commit
* merge
* rebase
* pull
  + gets new commits from the remote and updates the remote branches
* others…

These all move branches BUT only as a side-effect of creating/pulling new commits… what if there is an operation that explicitly just moves a branch? That is **reset**

The first step of rest is that it moves a branch (generally the current branch that ‘head’ is pointing to) and moves it to a new commit. Head points to the same branch (master, say) but it comes along for the ride as the branch looks at a new commit.

**Git reset –hard**  will copy data from the new current-commit into the working area and index area.

**Git Reset –mixed** will copy data from the new current commit to the index BUT won’t touchyour current working area. This is the default option

**Git reset –soft** doesn’t touch the other areas. It just moves the branch and doesn’t touch the index or working area.

Reset moves the current branch and optionally copies data from the repository to the other areas.

This module will be looking deeper into the use-cases of the powerful reset command. Let’s look at a few practical examples.

**A Real Git Reset Example:**

First let’s make a few changes to the working area and index.

We have the current commit (master). The head pointer is pointing at master and it’s looking at our current commit. Now, let’s suppose we want to add something to a file. Add the line to your file. So menu is changed in the working area.

Now add it to stage it to the index area.

Now commit it. Git creates a NEW commit, and it moves current branch (and head pointer of course) up to THAT commit.

NOW let’s add a new line to another file into the working area. Git add and git commit it. Awesome, now we’ve got a new commit.

Now suppose we need to go back to the initial commit and change that FIRST change that we made. **Unfortunately we have two commits that reference this bad commit we want to remove.** This is the strength of git reset. We can go back in time and fix things.

**Git reset --hard <commit hash>**

**→** takes you back in time and moves master back to the previous commit

→ head follows it

→ since it’s a hard reset, the old versions of the files are moved to both the working directory and the index

→ the two commits that have no branch pointing to them will eventually be garbage that is removed

**More Git Reset Examples:**

So we know git reset is about moving the header to other commits throughout the history of commits. Last time we saw a hard reset revert an entire project to the state it was two commits back in time. There are many more reasons to use git reset.

Starting from a clean status – suppose we edit a file, stage the edit with git add, and THEN change out mind and we want to clean the stage file? We want these in our working area BUT we want them removed from the staging area. **git rm –cached** is a way to do this, however git recommends using reset for this.

**Git reset HEAD** will move the current branch to be pointing at the same branch that head is. The second step of a regular reset will move data TO THE LEFT from the repository to the index but NOT to the working area. And the result is that it unstages all of the changes!

OH I GET IT! It’s resetting to head (current commit in repo) which is the same branch that you’re currently on so the SECOND part of a default **git reset** is to sync up the repo and the staging area (index) !!! this leaves the working area untouched.

**Git reset –hard HEAD** does the same thing BUT it resets the working area as well! It’s a popular command, but a destructive one (be careful). It says “please reset me to the current commit.

**More Tools:**

**The Stash: (very powerful!)**

The stash is the fourth area – the commands we’ve used so far have affected the other three but NOT the stash. If you want something to happen in the stash you have to use **git stash**. The stash is all yours, even moreso than the working area!

Suppose we create a new file in our working area.

Let’s stage it to the index (git add)

Let’s also edit an existing file.

Now imagine that you get interrupted during that work, but you don’t want to lose your half-baked working area! You can store the current status with:

**git stash –include-untracked**

**→** this will store your current state so that you can work on other things!

→ copies the working area AND the index and stashes it!

→ THEN it checks out the current commit again!

**Git stash –include-unstracked** will stash your current state and set you back to the clean state of the current commit!

**Git stash list -** lists all of your stashes

So the logical question now is “how do I restore the state from my git stash?”

→ our half-done work stays in the stash and we go off to do some other work

**git stash apply <label from list>**

→ and there we are! All of our data is back in the working area and index as it was when we stashed it for us to continue with! Very powerful! No more need to have multiple git directories!!!

**Solving Git Conflicts:**

Merge conflicts are very common and can be very annoying. In real life there are ‘merge conflicts’ and other conflicts all of the time.

For completeness here is an example of how to fix a common one as it gets you thinking about the four-areas.

Let’s create a branch called “Tomato” and add the line “tomato” to our file. Add it and commit it.

Now let’s switch back to the master branch, and modify the recipe in there as well by adding “Pepper” and commit THIS one.

Now let’s try and merge the tomato branch into our master branch…. Uh oh, there’s a conflict! If you ask for the status you’ll see a ‘both modified’ conflict error. Both sides added a line, one merged, now the other cannot. Git can’t decide which change should override the other or if BOTH should be there….

This will modify the file in the working-area with a diff-looking section.

Use **ls -la** to see some of the hidden files. Notice that there are several text files that are related to the attempted merge. Let’s cat them out.

There are merge guards

The merge head file is a reference (Temporary). Only lasts as long as needed to create the merge. It points at the branch (commit), but now we’re frozen in the middle of the merge so we can abort OR fix the conflicts and move forward.

Some conflicts are complicated but this is easy.

If we check out the edited file we see that git clearly marked the diffs within the to-be-edited file. We can manually edit this file and decide what it should look like (often combining the two versions!)

‘git status’ will still see a problem so we have to tell the index (Staging area) that we’ve fixed the conflcit! ‘git add’ the file back in there and we’ll be good now.

All we have to do is tell git to commit now and that’s how we solved it manually. That’s it. It’s interesting following git through the idea of the four-areas!

**Working with paths:**

Commits are pretty coarse in a way. It’s a snapshot of the whole project at any time. Sometimes you want to work with a single file or directory.

Let’s say we have:

menu.txt

readme.txt

Let’s make both files ALL UPPERCASE for some reason.

Stage the changes with ‘git add \*’

Now suppose we like the change to the readme but we’re not sure about the menu file just yet. You can do a HEAD reset, but we can also do that on a single file!

**Git reset HEAD menu.txt**

Copies data from the current commit to the index. Normally such a reset would copy everything over but this time only the menu file gets copied! Check it out with git status!

Let’s go one step farther with a HARD head reset on that file but no that doesn’t work. Git won’t do a hard head reset with a path. That’s just the way it is :(

To revert a single file without touching other files is to use checkout!

g**it checkout HEAD menu.txt →** checkout will not move head, just pull the current info from the repository into the working area and index!!!

Now the only file we have left staged is the readme and that’s what we want, so yes, we can finally commit it! We used the same commands in the past with different options and go different options! We’re getting better now!

**Git is a toolbox (concept):**

Tools are specialized. Each tool has features but they can all do many jobs! Different tools can also do SIMILAR jobs if you’re clever about it.

Git won’t give you commands like “unstage” because there’s already man ways to do that. You MUST understand the four working areas in order to use git properly because of this.

This is a very unique and linux-like approach, (made by Torvalds in fact) in this way. Look at your index. You can do so many things with just a few commands! Git good at them! Module completed. Keep the four working areas in-mind at all times!

**History Exploring the Past (of your project):**

This module is all about making sense of how your history works.

Git feels like a surgeon in this sense. You do EXACTLY what you want and there’s no more tools that have to deal with guessing.

We’re going to focus mostly on the repository! Hooray!

In these two models we’re going to focus a little bit more on the commands here than we had previously. You index will be expanded for sure.

**A commit by any other name:**

You can look at ‘**git log**’ for a history of commits, but it’s harder to make sense of what’s really happening in there.

**Git log –graph** will display git log in a new graph form

**git log –decorate** will make it look better

**git log –decorate –oneline** will condense it

Let’s say that we want to see which changes were introduced by the current commit!

**git show <hash>** will show you a much more technical explanation of what a commit changed!

**Git show <branch>** also does this.

**Git show HEAD^** will show you the parent commit to head! You can climb this way.

**Git show HEAD ^^** will show you the parent of the parent! This is fine for as far back until you have commits with multiple parents/comits before them!

**Git show HEAD^2^2** will do the second parent of TWO commits back

**Git show HEAD(9’1 month ago’)** will show you the commit from one month ago! Very cool!

**History Forensics:**

Some tools to figure out where your changes happened.

**git blame <filepath>** will show you the changed lines and the LATEST commit that changed those files (very cool)!

**git diff HEAD HEAD^2**  will show you the change of the current commit vs two commits earlier!

**git diff <branch> <branch2>** will diff two branches before commits!

**Browsing the Log:**

Git log is insanely powerful. You don’t have to memorize but just know about it. Feel free to not memorize and just google as you go but here’s a few very useful ones:

**Git log –patch** will give you a DETAILED git log that’s color-coded.

**Git log –grep apples –oneline** will only show log info for lines containing “apples”

**Git log -Gapples –patch** will show lines impacted by changes that either added or deleted ‘apples’ in the line.

Git log can also visualize a specific range of commits.

**Git log -3 –online** limits range to last three commits

The point of this module is that you have no know that these powers EXIT. Don’t just run to github crying whenever you need to find a change, use git log and google the options if you don’t remember them. Just know that they exist!

Module Completed. Tested it all out. Makes sense. Let’s go.

**History: Fixing Mistakes:**

In the last module we’ve seen a few commands about EXPLORING the project history, but now let’s edit.

Rebasing changes history technically, but let’s look at some more-advanced surgical tools. They become extremely useful for fixing mistakes quickly.

First of all… you should NEVER rebase shared commits! Rebase changes HISTORY. IT copies all commits to new commits. There are entirely different objects in the database and a lot of confusion can happen. Only use rebase for yourself!

**Fixing the latest commit:**

This is the easiest as it’s hardly going back in time, right?

‘git branch’ - okay we’re on master branch. Let’s add a new line to “menu.txt”

Stage the change…

Commit the index…..

‘git log’ - there, it’s in the log now. Now after committing we realize we didn’t quite finish the job! We need to edit ‘recipes.txt’ !

Instead of making a new commit let’s create the changes and stage them.

Now we’re in the process of fixing the problem, but creating another git is impropper. This is an inconsistent state! We don’t want a bad commit ANYWHERE as it can cause problems! Let’s add the changes to the latest commit instead of making a new one!

**git commit –ammend** will change the latest commit INSTEAD of creating a whole new commit (impropper)

Now the current branch and head are moved to the new commit and the old version of it is eventually turned to garbage that is removed later on.

**Interactive Rebases:**

Suppose we’ve made mistakes in the past and done two commits past it! (oh no!). We can’t use ‘git ammend’.

Use **git blame** to see when these lines were added…. Okay, we’ve located the commit. Now let’s look at the log to see when they happened.. The two mistakes were made VERY long ago but MUST be fixed throughout the whole thing! But it’s one of the very first commits. So how can we possibly fix this?

Look at the position of the remote branches in the log. We’ve located two bad commits, so let’s fix it the later of the two so-far because the earlier one is shared already (don’t do that).

First let’s create the fixed file with the fix.

Done, let’s stage it and commit it.

**Git log –graph –decorate** will show us where we went wrong.  
 HERE IS A POWERFUL COMMAND REBASE.

**git rebase -i origin/master** will bring you into a list of commits in the opposite of local order. Do not get confused. Now time to interractively fix it. We can force our commit into the one that was wrong in the past!

you’ll see:

pick <hash> <description>

pick <hash> <description>

etc.. etc……

So let’s make some changes! ‘pick’ means “don’t change anything”

Say we want to change the description to one while we’re here. Change the command from ‘pick’ to ‘reword’ and change the description up top.

Now ever see two commits that should be one commit? Make the command ‘squash’ on the higher ones to squash it down!

Exit and save and it fires it off! First thing it does is ask you for the new commit of the first squash!

The pick goes automatically.

Now git stops at the conflict that we fixed by-hand. Now what we’re doing is getting rid of that merge by squashing everything into a single commit instead. Now we have to fix the conflict instead. Once again, we edit the ‘bad’ file, solve the conflict, and run **git add <file>** followed by **git rebase –continue** to squash the commits together.

Then it’ll do the rest of the picks in order safely. This is awesome as was extremely powerful. Git rebase with interactive is extremely powerful.

“It’s like having an ‘undo’ operation always available!”

Most people will do a rebase before you push up to origin! Amazing!

Do lots of work…

Squash messages together…

Reword descriptions….

THEN finally push.

Given the power i’ll test it out now… HOLY COW I DID IT! You just have to do **git rebase -i <branchToStartWith>** and it’s in reverse time order!

**The Reflog:**

Whenever we change history – git has to copy all new commits from the old commits to totally change history. They’re usually unreachable! So they stay in the object DB for a while until git decides to garbage collect them.

What if we change our mind and we need one back? What if we delete a commit by mistake and need to recover them? They’re in the object DB but we don’t have their hashes anymore. There is a very easy way to do this.

Let’s checkout some branches. Git logged these movements into the ‘reflog’ , the reference log.

**Git reflog HEAD** will tell you all of the changes that happen to HEAD (pointer to current branch, remember).

This is like ‘history’ but for git only. Very Cool! But it’s not meaningful! Look at the numbers though! We can use this with **git show**

**git show HEAD@(15)** will show detailed information for what was done at change 15 for the current reflog (which we just viewed above with git reflog HEAD). Very cool.

Hopeflly it makes you feel a bit safer when destroying bits of history.

g**it reflog *refs/*heads/master** will show you when the local master branch was created. Hopefully that will make you feel safer because now you are able to recover and find your old hashes thanks to the reflog!

**Reverting Commits:**

Changing the history is something that we should be careful about – let’s find a non-destructive command without changingggg any commit… just creating a NEW commit!

Let’s go to another branch and log at the log…

We find a commit a little back that breaks the rules of the project (bad). We fixed one of these with an interactive rebase, but THIS errror has been shared so we want to fix this non-destructively. How can we do this SAFELY?

Let’s remove the error altogether.

The first way is the nuclear option – a huge interractive rebase way back in time and remove the incorrect commit. We can do this, but this has been shared, and shared history changing is very bad.

The proper way to do it is to keep this history but fix it ‘later on’. Git can do this automatically.

**git revert <bad commit’s hash id>** creates a brand new commit that is just changes that are the opposite of all of the changes from the commit whose hash you gave. It’s absolutely amazing. There is a new commit. The badness stays in history, but it’s better than editing a shared commit!

All that revert can do is writing the opposite of data. It cannot revert the STRUCTURE of changes. It cannot remove the merge commit itself. That’s still there. It mayy cause some confusion later on, so be careful. Be careful when you revert merges. It’s a special case, so be smart about it. Dont’ think about a revert as a magic undo – reset is better for that…. Revert is more-narrow. It just makes a new commit with new data that’s the opposite of current data.

Module Complete. You are a git history surgeon now.

**git commit –ammend** change current commit

**git rebase -i** change history interractively

**git reflog** see history of commits included destroyed ones you can check these out

**git revert HEAD** safely create a new commit that’s the opposite of the commit you point it to (in this example HEAD is whatever branch you have checked out)

**Finding Your Workflow:**

**Workflows and Pain:**

Final module of mastering git. So far we’ve made the assumption that you’re working alone on one repository and not working with other repos…. This obviously isn’t always true.

**You need a distubution model:**

How many repos do you have?

Who can access them?

Which branches do you have and what do you use them for?

Which branch is for releases?

Which branch is for combining work with other developers?

Do you merge code or do you rebase it?

These three things together define a distributed workflow. How to use git in-practice on a real working project.

THIS is a module about git workflow patterns. In a way you can think of this module as git workflow patterns.

You must know what a remote is.

You must know what pushing and pulling from a remote is.

**Distribution Models:**

**Peer to Peer Model:**

If you’re a dev in a team of three, you all start with the same, and each of the developers can see the other developer’s stuff as remotes. You can pull changes in and changes spread around like that. It’s simple but it’s not neccesarily easy as no repo is more-important than any of the others and it becomes a challenge to release the project. You need to decide which repo to release from and decide which one is right. It is way harder with even more developers.

**Blessed Repo and Centralized Model:**

You might say one repo is a “blessed” repo – accessible to everyone in the repo for pulling and pushing. You can push data there. Most devs call this “origin” (ohhhh!). It’s often just a repository with no working area or index – it’s just used to share data. Think about this as the “master” branch of sorts.

In this model you only care about the ‘blessed’ repo – or the official state of the project. The developer who has new commits must push them to the blessed repo. Most developers cannot write to this blessed repo.

**Branching Models:**

After your distribution model you need some policy for dealing with branches.

Stable vs unstable branches:

A stable branch always keeps something STABLE in the latest branch that’s always ready for depltoy.

Un unstable branch means that the tip of the commits may be not-working but you have the latest stable one deployed to prod.

Almost every project has a main branch that you use for everything together (think ‘MASTER’) and tend to stay aligned into the main branch and flow back into said main-branch. It’s usually called the master but it’s also called the *integration branch.* Some people call it the main branch.

Is it stable or unstable? This depends on your project. Most people prefer it being stable. Mostly everyone is working on it and nobody likes working on an unstable codebase. In practice most projects aim for a mostly-stable integration branch… Jenkins attempts to do this for you. Get good with jenkins.

Another important question is “What branch do you release software from”? Many projects release from integration branches! Other projects have a ‘release’ branch. Consider using a release branch as well as the master branch.

Sometimes you branch off and have different strings of branches for developing each features. When a ‘FEATURE BRANCH’ is finished it is merged into the master integration branch. This allows multiple branches of feature branches and the integration branch ONLY has fully finished features! Feature branches with an integration branch is very important!

An example is that you can ‘cherry-pick’ commits from feature-branch2 for feature-branch1, but this doesn’t quite work perfectly. Just use a THIRD branch (shared feature branch) in both and merge it into both the first and second branch!!! This is very common in development while you have a bugfix during development.

**Constraints:**

Catchall name for stuff that doesn’t fall into the other models. There are many many possibilities here but there are some examples that you should consider and keep in mind.

* Do you merge or do you rebase when a feature branch is ready to come in?
  + Be consistent no matter what you choose
* Who can do what to which branches?
  + Do all developers have access to all branches?
    - Do you limit it by feature-branch?
* How do you handle red builds?
  + Freeze all pushes if there is a ‘red build’ (bad build)
  + do NOT let people violate this constraint
* **git hooks**
  + git hooks run something whenever ‘something’ happens.
  + If the build is ‘red’ you should be able to stop all pushes
* what should you do before merges?
  + Some projects prefer squashing everything to one single commit!
* Etc etc etc -… it’s important that you build your git workflow!

So remember we need:

* distributed workflow
* branch workflow
* constraints for workflows (hooks etc)

-...for a successful git project!

**Gitflow:**

Gitflow is a way of working with git. IT’s based on a centralized distribution module. It also encourages developers to exchange data WITH peer to peer development.

The branch model divides branches into stable branches and unstable branches with an integration branch between them. Using the feature branch model is recommended. Master MUST be stable here – release branches are whenever you deploy a master branch. Thing, when you package a new master.

Git flow believes in merging all the time and only rebasing if there’s an emergency.

Git flow also gives you the naming conventions! It’s very tightly defined and it has many advantages AND there are extensions on the internet! You can find premade hooks online!

“Dont’ just use gitlow” - the instructor advises. There’s nothing ‘wrong’ with it and it includes sound versioning practices, but don’t just grab it and use a precanned workflow. Change it to what you need!

Context. Is. Everyhing.

**Coming up with your own workflow:**

This is more art than science, keep in mind. Don’t just sit down and design a workflow. It makes things overcomplicated and doesn’t address the right problems. Build it AS YOU GO. It helps but likely will end up having been a waste of time. Build the workflow as you go.

GROW your workflow. Start small. Details don’t matter too too much.

Keep it simple. You will rarely regret doing that.

**REVIEW:**

The working area is split up into the working area, the index, the repository, and the stash.

Most commands can be understood in “how they move data across the area” and “does it change the repository”?

We learned how to work on the history with git log, git diff, and git blame.

We learned how to fix history with things like git rebase, git reset, and using the reflog and recovering from our desctructive mistakes.

Finally we talked about workflows and git hooks.