Git

Table of Contents

Git is a distributed version control system which allows teams and individuals to manage changes to a project over time.

What is the problem that git tries to solve? If you've ever tried collaborating on a single Word document with multiple colleagues prior to Microsoft O365 Online, you may already have experienced the problem that GIT solves. In order to all colleagues to be able to make edits to the document, you may have emailed updates or changes to one another and tried to maintain a "latest version". Keeping track of the latest version was frustrating to say the least. You may have also tried to keep the document in a shared directory or mapped volume, but this would often cause frustration with "locked" files and the ability for only a single person to be editing the document at a time. Git solves this problem by managing how changes are applied to a central repository of data.

In this session we are going to go over common git tasks, terms and workflows using a sample project being operated on by multiple people. The goal of our project is to write a collection of scripts that can be used by our team to automate a lot of our daily tasks.

Getting Started

Getting access to git.target.com

Depending on your role, you may already have access to GitHub Enterprise. You can verify by attempting to log in to https://git.target.com. If you do not have access to GitHub Enterprise, you can request access by following the instructions at
https://wiki.target.com/tgtwiki/index.php/GitHub_Enterprise#Getting_Access
or you can still complete all of the activities with a free GitHub.com account.

Next we need to set up two-factor authentication for our account which is required on git.target.com. Regardless of whether you are using GitHub Enterprise (git.target.com) or GitHub.com, you should set up two-factor authentication. GitHub has a full help document on two-factor authentication for your account. There is also an abbreviated instructions list on the Target wiki.

Installing Git

Mac

Git on the Mac is installed alongside the XCode Command Line Tools. These tools can be installed by opening the Terminal.app and typing "xcode-select --install" and following the prompts.

Windows

The best way to install Git on windows is to go to the official git download page and run the installer.
https://git-scm.com/download/win

Going forward, we will be doing most of our exercises in Terminal.app on Mac and GitBash on Windows. For the rest of this document, Terminal.app and Windows will collectively be referred to as "CLI", or command line interface.

Configuring Git

SSH Key Auth

Every time we want to download code changes from a private git repo or push new code changes to any git repo we need to authenticate to the service that is hosting our git repo. GitHub Enterprise offers two ways to connect to these services: https and git+ssh. Https uses standard basic authentication to authenticate a user which means that we would be asked to provide our username and password each time we connect to the service hosting our git repo. Git+ssh on the other hand, uses ssh key pair authentication. Your public key is stored in GitHub Enterprise and your private key is added to your local key agent. With your private key in your key agent, git will automatically use these keys to verify your identity and save you from having to type your username and password each time.

Instructions for creating and adding SSH keys to GitHub Enterprise can be found at https://help.github.com/en/enterprise/2.18/user/authenticating-to-github/connecting-to-github-with-ssh.

Local Configuration

If this is the first time you've used Git on your computer, you will want to set some configuration items. All committed changes to files in a git repository are tracked and able to be traced back to the person who made the change. In order for this to happen, git needs to be told who you are. The first configuration that you should set is your name and email address. We can also change the default text editor used for many git commands.

Set your username
```
git config --global user.name "John Doe"
```

Set your email address
```
git config --global user.email johndoe@example.com
```

Change the default editor for git (optional. Defaults to vim)
```
git config --global core.editor "pico"
```

We'll talk more about this later, but for our exercises we will want to visualize our changes. Copy and paste the following to allow us to do that.
```
git config --global alias.graph "log --graph --abbrev-commit --decorate --format=format:'%C(bold
blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold
green)(%ar)%C(reset)%C(auto)%d%C(reset)%n''          %C(white)%s%C(reset) %C(dim white)-
%an%C(reset)' --all"
```

As you may be guessing right now, setting this configuration allows you to put in any value for your identity without verification. Without some added protections, it is possible to impersonate someone else and also have someone else impersonate you. Most of the time this concern is small considering we trust ourselves and our coworkers not to impersonate another. There are times, however, where we will want an additional level of trust so that others know that any commit with our name and email attached actually did come from use. The full setup is outside the scope of this lesson, but instructions can be found at https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work. Be sure to read the last "**Everyone Must Sign**" section in that document before proceeding.

Creating your first Git repository

Git keeps track of all of the changes in a particular directory. In order to create our first git repository, we will need to first create a directory that will contain all of our project files. In your Documents folder, create a directory name “my-first-git-repo”.

Now open the Terminal app on your Mac or PowerShell on Windows and go into your new directory by typing “cd ~/Documents/my-first-git-repo”. To create a clean Git repository, just type “git init” from within that new directory.

To clarify some terminology, Git repository does not exclusively refer to a project on a hosting service such as git.target.com, github.com or others. Git repositories can live anywhere and can have many copies or "clones" of themselves.

Adding new files to version control

Now that you have an empty git repository, run “git status” in that directory to see some statistics on your repo. In the output you should see that you are on the master branch, a note that you have not made any commits and that you have nothing to commit. All good software projects contain a file that describes the project along with possibly showing some documentation on how to use the project. This file lives in the root of the repo and is named “README.md”. Create that file in your directory add a line with the contents “# My First Git Repo”. The file is a Markdown file and this line uses the markdown syntax to add a heading to our file.

Another file that we will want to add to all of our git repos is a ".gitignore" file. These files list everything that git should exclude from having their changes tracked. Temporary files & folder, files with secret information like username & passwords and IDE configuration files are all good candidates to be ignored. Create a file called ".gitignore" and add a single line with the contents ".env". A dot env file will often contain usernames and passwords used to connect to other services and should not be included in the repo for the service.

With our new file created we then tell git that we want to start tracking changes to this file. We do this by first “staging” the file. Staging a file means that we are preparing to commit a change to our repository. Back in your CLI, type `git add README.md .gitignore`. If we type `git status` again, we can see that we are now also shown a list of files that changed and been “staged” for commit.

Finally, we can commit those changes to our branch. When committing the changes to our branch, we also want to specify a message that describes the changes in the commit. Since this is our very first commit, we can describe the commit as simply “initial commit”. The command to commit the change would then be `git commit -m "initial commit"`.

Sharing Code

We have now created a repository on our local computer but have not associated it with a repository on a remote hosting service. What this means is that the repository on our local computer is isolated and not able to include changes made by others. It will also be difficult to continue working on our project if we use multiple computers. While having the local repo is still useful considering we can still track our own changes and revert to previous versions of our changes, it might be more beneficial to link our local repository to a repository which others also have access.

Go to https://git.target.com or https://github.com and sign in. After signing in you should see a list of your repositories on the left. In that section there should be a "New" button to allow you to create a repository. Give your repository a name and set whether or not it will be a public or private repository. Since we already have a local repository that we plan to sync with the new repository, we will NOT Initialize this repository with a README, nor will we add a ".gitignore" file or "license" file.

We now have a new remote repository at the URL shown in the completion page. In order to sync up our local and remote repositories, we need to make a change to our local repository. In our local repository, we need to set a setting stating that our local repository is a clone of our remote repository and we should use our remote repository as our source of truth when it comes to changes. We do this because our remote is the repository to which everyone has access. The "source of truth" repository is typically referenced via the "origin" name. Back in our terminal app, we can set our origin repository with the command:

```
git remote add origin <remote-url>
```

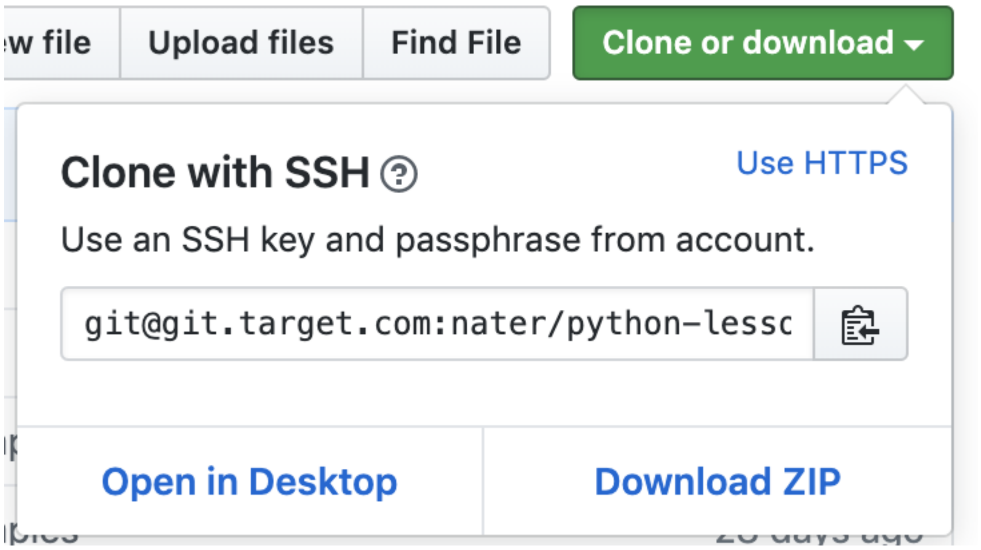
Now that we have an origin remote, we can take all of the commits that have been created locally and push them to our origin remote with:

```
git push -u origin master
```

Now refresh the webpage on git.target or github and you should see all of your commits.

Cloning a different Git repository

Throughout your career, it will certainly be necessary to start working on a git repository that already exists. The process of creating a local repository based on an existing remote repository is called "cloning". To clone a repository, you will need to obtain the URL of the remote repository. For GitHub Enterprise and GitHub, that link is available on the main page of the project. When you click on the “Clone or download” button, you will be presented with the URL. Since we have our SSH keys set up, we will want to make sure we are using the SSH URL. If the modal includes "Clone with SSH", you're set. If not, you should see a link labeled "Use SSH" to switch to that URL.



Back in your CLI, you will use the `cd` command to change directories to where you want to save the local repository on your computer and then copy the URL from the website that starts with "git://" and use the `git clone <url>` to create the local repository.

Working with Remotes

A quick note on commit hashes. In all of the examples below, you will see a commit hash (something like **6dabe5a**). Commit hashes are uniquely generated. As you go through the exercises, your commit hashes will be different so you'll need to make certain that if you are copying and pasting commands, that you are also updating your hashes to match your repo.

Pushing changes to remote

With our local repositories now set to track our remote repository, we can start to make changes locally and "push" those changes up to our remote. Let's make a small change our README.md file and add a description. Under the title, add a note "This is where the project description would go". Just like before, we will save that file, stage the file with `git add README.md` and then commit that change with `git commit -m "update readme"`.

Now if we were to type `git log`, we should see all of our commits and you will notice that there is a new entry for our commit that we just made. This entry will also have a descriptor next to it marked "(HEAD -> master)". Below that you should see another commit with a descriptor "(origin/master, origin/HEAD)". This tells us that our local repository is ahead of our remote repository, or in other words, we have changes locally that do not exist on our remote repository. In order to put those changes on our remote repository, we can type `git push`.

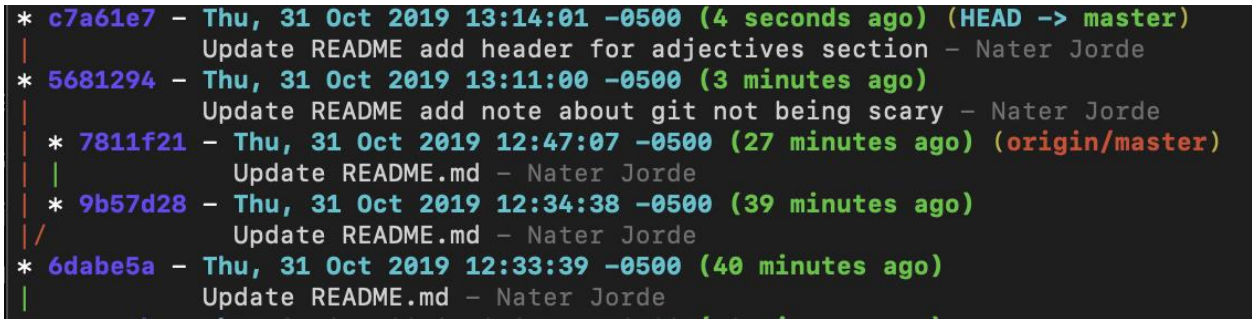
Pulling changes from remote

Both Git Enterprise and GitHub allow us to make changes to code directly in our web browser. Let's use this function to make a change to our README.md and add another line to the end of the file. Open the project in the web browser and click on the README.md file. Click the pencil icon and add "Git is good" to the end of the file. At the bottom of the page, click the "Commit changes" button to accept the default "Update README.md" commit message. Now go back into your CLI on your computer. If you know that you will want to download all changes on the remote repository, you can simply type `git pull` to download and apply all changes from the remote repository. Finally, if we type `git log` again, we should see all of the commits from our remote repository downloaded to our local repository.

In some cases, you may first want to check the changes on the remote before applying them locally. First, you'll need to download the record of all changes made on the remote with `git fetch`. Now you can see a list of all of the commits that differ by typing `git log master..origin/master` and if you want to see all of the changed lines, `git diff master..origin/master`.

Managing conflicts

At some point in our development, the commits to our local and remote branches are likely to diverge. Basically, you may be making some changes locally and one of your coworkers has made different changes and pushed them to the remote repository. In the git log shown below we can see that the origin and local repos were synced up at commit **6dabe5a** but the origin repo includes 2 commits that are not in our local repo and the local repo has two commit that is not in the remote. In addition to this, the commits are both updating the same README file and have likely edited the same line. Git will be unable to automatically resolve these differences, so we will need to do it manually. Git provides us with two ways to manage these conflicts, merging and rebasing.



```
* c7a61e7 - Thu, 31 Oct 2019 13:14:01 -0500 (4 seconds ago) (HEAD -> master)
|           Update README add header for adjectives section - Nater Jorde
* 5681294 - Thu, 31 Oct 2019 13:11:00 -0500 (3 minutes ago)
|           Update README add note about git not being scary - Nater Jorde
| * 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (27 minutes ago) (origin/master)
| |           Update README.md - Nater Jorde
| * 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (39 minutes ago)
|/            Update README.md - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (40 minutes ago)
|           Update README.md - Nater Jorde
```

Merging

We currently have some changes on our remote repository that are not included in our local repository and vice versa. If we try to do a `git pull` we are going to get a message stating that an automatic merge has failed. Opening the README.md file will show us where that automatic merge was unable to complete with lines denoted by `<<<<<<< HEAD`, `=======` and `>>>>>>> <hash>` with our change between that text. Our goal is to replace all of the code between `<<<<<<< HEAD` and `>>>>>>> <hash>` with the code as it should appear in our final version. Sometimes this will involve selecting some code and deleting other code. Somethings it might involve selecting parts from both sections. Using a 3rd party merge tool may help us make that choice. One option that works on Windows, Mac and Linux is P4Merge. This particular merge tool will show us what a particular file looks like before the conflict, each version and a final version.

We can configure git to use the P4Merge visual merge tool by typing the following command.

```
git config --global merge.tool p4mergetool
git config --global mergetool.p4mergetool.trustExitCode false
git config --global mergetool.keepBackup false
git config --global diff.tool p4mergetool
```

To setup the p4mergetool command on the Mac, use:

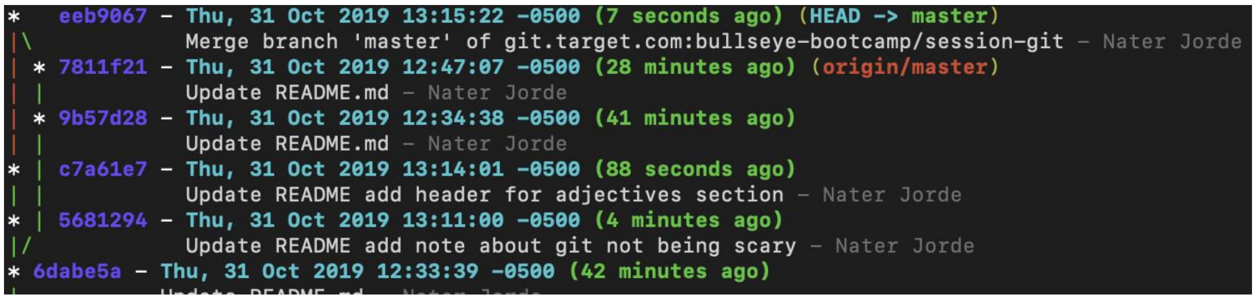
```
git config --global mergetool.p4mergetool.cmd
"/Applications/p4merge.app/Contents/Resources/launchp4merge \$PWD/\$BASE \$PWD/\$REMOTE
\$PWD/\$LOCAL \$PWD/\$MERGED"
git config --global difftool.p4mergetool.cmd
"/Applications/p4merge.app/Contents/Resources/launchp4merge \$LOCAL \$REMOTE"
```

To setup the p4mergetool command on the Windows, use:

```
git config --global mergetool.p4mergetool.cmd "C:\Program Files\Perforce\p4merge.exe"
git config --global difftool.p4mergetool.cmd "C:\Program Files\Perforce\p4merge.exe"
```

Once the default merge and tool is set we just need to type `git mergetool` to open the editor and resolve our conflicts. After we've resolved the conflicts and saved the new file, we just need to type `git commit` to apply the changes. If we don't specify a message, a template message will appear filling in some details about are merge conflict.

In order to see what happened, let's use the alias we set up earlier and type `git graph` in our CLI. We should see all of our previous local commits as well as a new "Merge commit". The merge commit is shown to have occurred last in our git log and it includes all of the changes necessary to sync up our local branch changes with the changes in the remote version. We can see that all of the remote commits still exist and so does our local commit with hashes of **5681294** & **c7a61e7**. At the very end of the log in a brand new commit with has **eeb9067** which is our merge commit that includes all of the changes necessary to resolve our conflict.

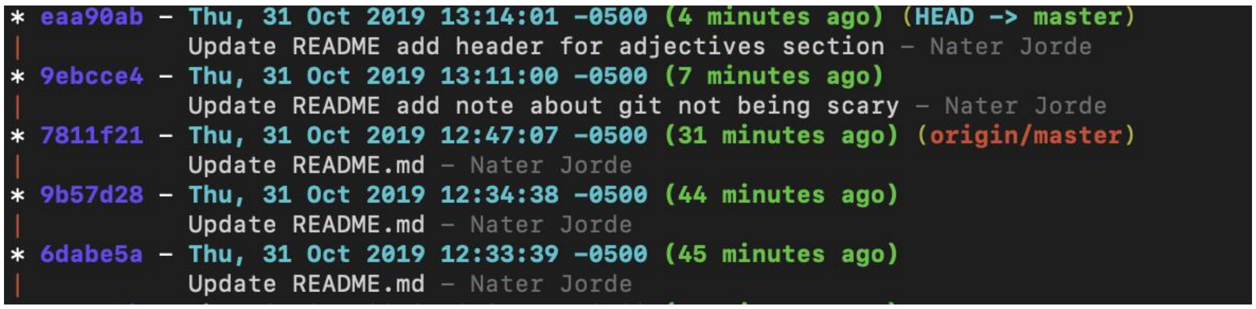


```
*   eeb9067 - Thu, 31 Oct 2019 13:15:22 -0500 (7 seconds ago) (HEAD -> master)
|\            Merge branch 'master' of git.target.com:bullseye-bootcamp/session-git - Nater Jorde
| * 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (28 minutes ago) (origin/master)
| |           Update README.md - Nater Jorde
| * 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (41 minutes ago)
| |           Update README.md - Nater Jorde
* | c7a61e7 - Thu, 31 Oct 2019 13:14:01 -0500 (88 seconds ago)
| |           Update README add header for adjectives section - Nater Jorde
* | 5681294 - Thu, 31 Oct 2019 13:11:00 -0500 (4 minutes ago)
|/            Update README add note about git not being scary - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (42 minutes ago)
```

Rebasing

Rebasing and Merging accomplish the same goal, which is to sync up changes in one repository (local or remote) with changes in the other. The way they accomplish this task is different. As we saw before, merging resolves changes at the end of the git history. Rebasing, however, applies the changes from the specified repository at the point where the two branches diverged or the specified commit. Since changes are applied to this point, all subsequent commits to the branch being rebased must be resolved for conflicts. We can start a rebase by typing `git rebase`. Since each commit will need to be resolved, rather than committing changes, we resolve the conflicts and use the `git rebase --continue` command to move on to the next commit.

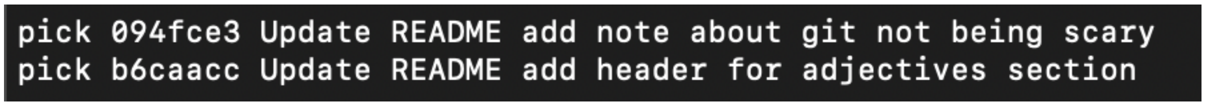
After the rebase is complete, you will notice that our history looks much cleaner. Rather than having a diverged branch that needs to be resolved, we have a single linear history. You will notice that all of the remote commits (**9b57d28** & **7811f21)** still exist as they were, however our original local commits (**d630f99** & **cd7986a**) have been removed and replaced with new commits with a completely different hashes (**9ebcce4** & **eaa90ab**) at the very end of our git history. From this we can see that rebasing is a destructive operation. It destroys the old commit and history in order to create a new commit and history.



```
* eaa90ab - Thu, 31 Oct 2019 13:14:01 -0500 (4 minutes ago) (HEAD -> master)
|           Update README add header for adjectives section - Nater Jorde
* 9ebcce4 - Thu, 31 Oct 2019 13:11:00 -0500 (7 minutes ago)
|           Update README add note about git not being scary - Nater Jorde
* 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (31 minutes ago) (origin/master)
|           Update README.md - Nater Jorde
* 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (44 minutes ago)
|           Update README.md - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (45 minutes ago)
|           Update README.md - Nater Jorde
```

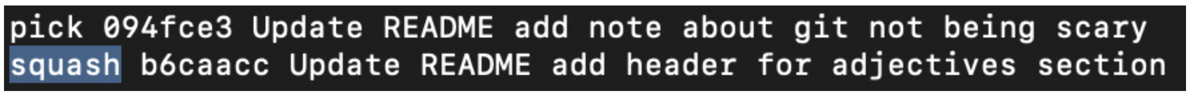
Interactive rebasing

In addition to create a new commit history with our local commits, git rebase also has an interactive mode that allows us to do much more. In our example, we have two commits that update the README. If we really wanted a perfectly clean commit history, we may want these two commits to be merged into a single commit. To start this process, we simply add the `-i` flag to the command: `git rebase -i`. We are first shown a list of all of the commits that we have made locally in the order that they were made. From here we can change the order as well as "squash" commits together.



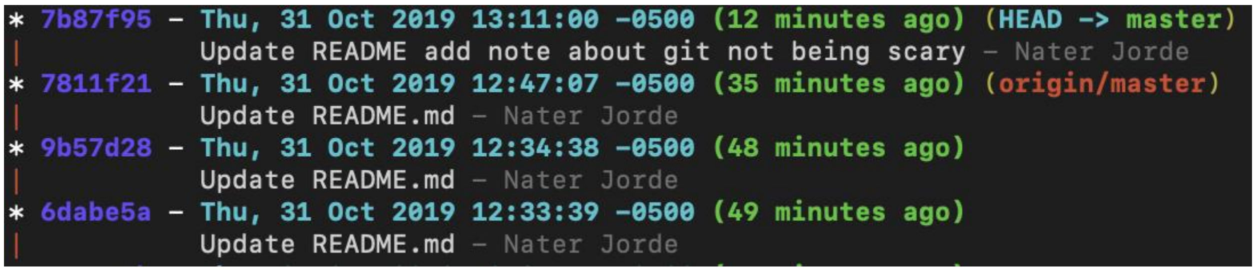
```
pick 094fce3 Update README add note about git not being scary
pick b6caacc Update README add header for adjectives section
```

Unlike git log, the commits are ordered from oldest on top to newest on the bottom. If we want to merge the two commits together, we need to edit the log so that the newer commit is squashed onto the older commit. In our example, we would change the word "pick" to "squash" for the commit with hash **b6caacc**. Once we save this file, git will start to attempt to resolve all of the issues in order one commit at a time. Since we had two commits, we will potentially need to resolve two conflicts.



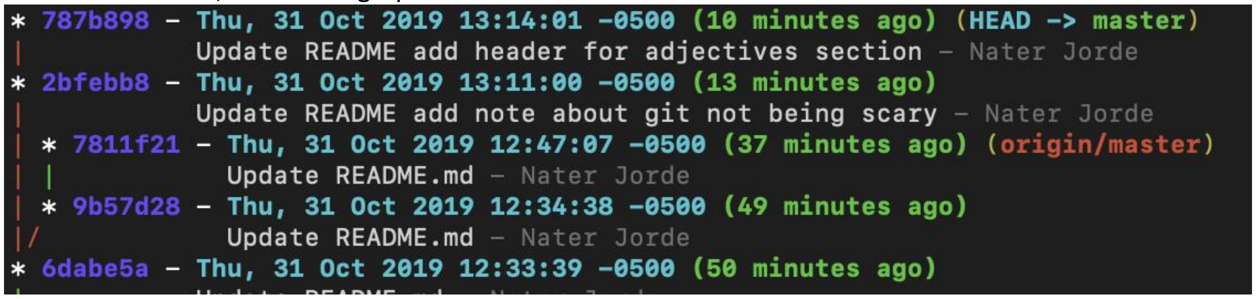
```
pick 094fce3 Update README add note about git not being scary
squash b6caacc Update README add header for adjectives section
```

Once the interactive merge is complete, our new history includes all of the remote commits, however our two local commits were merged into one commit and placed at the end of the git history.

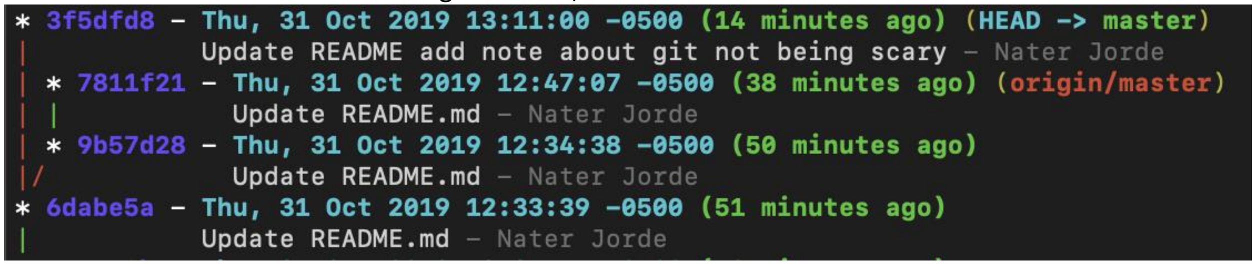

```
* 7b87f95 - Thu, 31 Oct 2019 13:11:00 -0500 (12 minutes ago) (HEAD -> master)
|           Update README add note about git not being scary - Nater Jorde
* 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (35 minutes ago) (origin/master)
|           Update README.md - Nater Jorde
* 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (48 minutes ago)
|           Update README.md - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (49 minutes ago)
|           Update README.md - Nater Jorde
```

Interactive rebasing can also be done on local changes only. This can be particularly useful prior to rebased onto a remote branch in order to reduce the number of commits that would need to be resolved. In order to accomplish this, we would simply need to specify a commit hash in our local history onto which the changes will be rebased. In our case, that will be the has at which our branches diverged `git rebase -i 6dabe5a`.

Before the rebase, we have a graph like this:


```
* 787b898 - Thu, 31 Oct 2019 13:14:01 -0500 (10 minutes ago) (HEAD -> master)
|           Update README add header for adjectives section - Nater Jorde
* 2bfebb8 - Thu, 31 Oct 2019 13:11:00 -0500 (13 minutes ago)
|           Update README add note about git not being scary - Nater Jorde
| * 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (37 minutes ago) (origin/master)
| |           Update README.md - Nater Jorde
| * 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (49 minutes ago)
|/            Update README.md - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (50 minutes ago)
```

After the rebase we still have a diverged branch, but it looks like this:


```
* 3f5dfd8 - Thu, 31 Oct 2019 13:11:00 -0500 (14 minutes ago) (HEAD -> master)
|           Update README add note about git not being scary - Nater Jorde
| * 7811f21 - Thu, 31 Oct 2019 12:47:07 -0500 (38 minutes ago) (origin/master)
| |           Update README.md - Nater Jorde
| * 9b57d28 - Thu, 31 Oct 2019 12:34:38 -0500 (50 minutes ago)
|/            Update README.md - Nater Jorde
* 6dabe5a - Thu, 31 Oct 2019 12:33:39 -0500 (51 minutes ago)
|           Update README.md - Nater Jorde
```

Considering we now have a single commit locally that diverges from the remote, doing a basic `git rebase` will be much easier as we will only have that single commit to worry about resolving.

Merging vs rebasing

So, if merging and rebasing do the exact same thing, why would you do one over the other? The main benefit of rebasing is to keep a clean git history. When working on large projects with multiple developers, keeping a clean history on your shared branches can be tremendously useful for tracking changes.

The downside to rebasing is that you are rewriting git history. There is a golden rule for rebasing and that is to never rebase a shared branch. If you rewrite history for a shared branch, any other user relying on that shared branch is going to have great troubles when attempting to pull changes to a branch with a completely different history from when it was originally pulled. If you are working on a shared branch and you wish to rebase, make certain that you are only changing history locally. Git does provide some safety nets for you if you attempt to push updates to a remote that would change history.

There are ways to use merging and still keep a clean history. Those methods are covered later in the git workflows section.

Git Branches

So far in our examples, all of our code has been updated on a single git branch called master. At some point we may start working on a particular update that we would like to keep isolated from the main master branch until all of the work is complete; we may be collaborating with another person on the update, the update make break something if any part of it is merged into the master branch before all work is complete, or we may be working on multiple stories/issues at the same time and would like a way to switch back and forth between contexts. For these cases, git provides us with the functionality to create multiple branches in our project.

Let’s use the last example mentioned and assume that we are working on multiple stories/issues at once. Our first story is to add a feature to our code and while we’re working on that, a high-profile bug comes in that needs to be fixed immediately. We want to be able to save all of the changes we've made and pause work on the story and easily switch over to fixing the bug without the possibility of accidentally incorporating the story code into the bug fix.

The first thing that we are going to do is to checkout our default branch (master in our case) and pull down all changes from remote to make certain we are up to date.
`get checkout master` and `git pull`. Next, we create a branch for our story work and since we currently have the master branch checked out, our new branch will be based on master. Creating a new branch can be done in one of two ways. We can first create the branch and then checkout that new branch to start working on the new story. This involves two commands: `git branch new-branch-name` and `git checkout new-branch-name`. The git checkout command, however, allows for an optional `-b` flag that will create a branch at the same time as checking it out. `git checkout -b new-branch-name`. At this point, you can edit, stage and commit files just as you have before.

While working on that new feature, you need to immediately start work on a bug that is present in the master branch. Before creating the new branch, we are going to want to tidy up what we are currently working on. If we have written code that is ready to have a commit created, we can simply stage and commit those files. If we have changes that we aren't quite ready to commit, we can stash those changes.

Next, we checkout the branch that our new branch should be based on. If our default branch is master we'll run the same `get checkout master` and `git pull` as before and then do the same `git checkout -b bugfix-branch-name` as before. Now we are ready to work on our bug fix without affecting any work being done in our other branch or having any work in our other branch affecting our bug fix. Since the branches are already created, we can switch back and forth between them by tidying up our current changes and then calling `git checkout <branch-name>` without the -b flag.

Git Stashing

At times you may be in the middle of some work and need to switch to something else or you may be making some changes or additions that would better be applied elsewhere. Stashing allows you to extract changes from the current project and "stash" them away for use later. The command to accomplish this is `git stash` and there are a lot of flags available for this command that we can use for specific purposes.

`git stash`
This will stash away all tracked and modified files. Only tracked files are stashed, so if you had created a new file or directory, the new file/folders will not be included in the stash. All stashes have a description message attached to them. By default, it includes the name of the branch and the most recent commit message.

`git stash -u`
Adding the -u flag tells git that it should include all of the untracked files/folder. This will include new files/folders in the stash.

`git stash save -u -m "desc of files that were stashed"`
Rather than relying on the most recent commit message, you can give your own description to better state the purpose for the changes to the files by using the save subcommand.

`git stash push -u -m "desc of stash" -- file1.txt file2.txt`
If you don't want to stash all changes, you can use the push subcommand and specify a list of files that should be included in this particular stash.

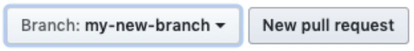
At any time, from any branch, you can list all of the stashes that you created with `git stash list`. In the output, you will see a list of all of your stashes along with a corresponding ID.

There are two options to bringing the changes stored in the stash back into the project. `git stash apply` will apply all of the changes and retain the actual stash. `git stash pop` will apply all of the changes, but delete the corresponding stash. By default, each option will apply the stash with ID of 0. If you wish to bring in a specific stash, you can specify it by id (i.e. `git stash apply 2`, `git stash pop 1`)

Pull Requests

One reason we may want to use branches is that our default branch is protected in our remote repository. Since releases to our product may be based on this default branch, we will want to make certain that no changes can be applied to that branch unless those changes have been reviewed by team members other than the one making the change. If we keep all of our changes in a branch we can make a request to pull those changes into a destination branch. While the git CLI does provide a mechanism by which we can generate a message to send to the owners of a project, typically we will use the GitHub Enterprise website to create our pull requests.

After all of your changes have been committed to your branch and pushed up to the remote repository, you should visit the remote repository page. In GitHub enterprise, you can switch between any branch using a dropdown menu on the repo's main page. To create a pull request, you can click the button directly next to the selected branch.



The pull request shows all changes that will be made to the target branch if the PR is merged. In GitHub Enterprise, you can request specific people to review, assign the PR to a maintainer, have a conversation about the changes and more. The pull request features will differ depending on which remote repository platform you are using, but documentation for GitHub Enterprise can be found here.

You and your team should also agree on some ground rules for completing pull requests and code reviews. Google recently released their guidelines for performing reviews and you may want to use that as a starting point for your own team.

Git Workflows

In addition to best practices for pull requests, your team or project may have explicit requirements for handling workflows resulting in merges to particular branches. There are four very popular workflows that we are going to cover in this session; Centralized, Feature Branch, Gitflow and Forking. Atlassian has written up an excellent comparison of each of these workflows.

Centralized

You may recognize the centralized workflow as it is the workflow that we've been using in most of our examples. Basically, this workflow uses a single branch that everyone commits to. This workflow may work great for small teams who need to quickly get code published. Hackathons, school projects, proofs of concepts all make this workflow an enticing option. The downside to this workflow is the inability to scale. The more developers you add to your project, the more difficult it is to maintain a clean project free of defects.

Feature Branch

The feature branch workflow is ideal for projects with scheduled releases of shipped products. Teams may wish to completely isolate a feature until it is ready to be “shipped”. Feature branches are often long running and frequently need to merge in changes from the default branch because of it. Feature branches may consist of multiple stories that make up the feature if the feature is large or complicated, though this is not required. Your “feature” branch may be a single story. In cloud based and continuous delivery (CD) models, the definition of “shippable” may simply be that code was added and all tests passed. The new code may be part of the larger feature that is not complete, but that is OK.

Gitflow

Gitflow is similar to feature branch workflow though it expands on and more strictly defines branches and their purposes. Since this workflow has strict definitions, it is great for large teams because new developers who have not worked in the workflow before have a lot of resources to learn the workflow. Being highly opinionated and structured also means this workflow is very well suited for maintaining good git histories of changes, a necessary process for certain compliance requirements.

Forking

This workflow is ideal for open source projects where a maintainer can solicit code changes from the community without needing to give write access to people they do not know. When followed strictly, another large benefit of this workflow is that the main project repository is kept very clean because only changes that have been approved to be included in a product are ever brought into the repository. This is accomplished by having all of the participants of the project create “forks” of the main repository and perform all of their work-in-progress commits completely separate from the main repository. Only approved pull requests are allowed to apply changes and in many cases, the branches in the main repository are restricted to never allow commits from being directly applied. Because things are kept clean, this workflow makes for an excellent choice for private projects as well as open sourced ones.

In all of the workflows other than the Centralized workflow, pull requests are used to restrict changes to the main branches. In GitHub enterprise, a repository can be setup to only allow "Squash Merges" when pull requests are approved. In these cases, if the feature branch includes multiple small commits, all of the commits will be merged into a single commit prior to being merged into the main branch. This helps keep the history clean without the need to rebase.

Git Tags

At some point you will want to mark some sort of an event on your repo such as when you create a release of your software product. Git provides this functionality via tags and allows for the creation of two types of tags: lightweight and annotated.

Annotated tags are full objects in the git database, they contain information about the person who created the tag, the datetime that the tag was created, and a message about the tag. Since they are full objects, they can also be signed for verification with GPG. Annotated tags can be created with `git tag -a <tagname> -m <message> [<commit hash>]`. If you do not specify a commit hash, the tag will be associated with the most recent commit on the branch that you have currently checked out.

Lightweight tags simply reference a commit and do not allow for providing any additional information. In general, annotated tags are preferred over lightweight tags. In addition to not providing additional information, lightweight tags are not listed when using `git describe` and not included when using `git push --follow-tags`. Lightweight tags can be created with `git tag <tagname> [<commit hash>]`. If you do not specify a commit hash, the tag will be associated with the most recent commit on the branch that you have currently checked out.

Tags are not included by default when pushing changes to a remote. In order to push a created tag to the remote you can either specify it by name to push only that tag with `git push <remote name> <tagname>` (i.e. `git push origin v1.0`) or push all tags with `git push <remote name> --tags` (i.e. `git push origin --tags`).

Tags can also be deleted but the process to do that will differ slightly if you are deleting them locally or on a remote. If you are just deleting a tag locally, you can use `git tag -d <tagname>` to delete the tag. Since you would need to push up a change to a remote in order to delete a tag remotely, you can accomplish deleting a remote tag with `git push <remote name> --delete <tagname>`.

Git History

Git doesn’t just allow us to keep a “latest” copy of a file, it also manages the entire history of changes to our files. Every change to every file is recorded via git commits and all of those commits are available to be inspected via the git history using the `git log` command. Git log includes MANY flags and options that we will find useful. Some flags that will help us to visualize git history are `--graph`, `--all` and `--format`.

Back in our CLI, we can copy and paste the following line

```
git config --global alias.graph2 "log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)-%an%C(reset)%C(auto)%d%C(reset)' --all"
```

This will create a git command alias that prints out our git branches in a condensed chart by typing `git graph2`.

Git Blame

When working on a project you will likely run into some code that is not immediately evident as to why it was written in the way it was, or there may be some questions on the business logic being used. As you may imagine, it can be useful to look up information on a particular line or block of code to help track down the answers to those questions. Git provides this functionality via the blame command.

Typing `git blame README.md` from within the root of our project directory will print out the contents of the README file and show us line by line the name of the person that last updated that particular line as well as a timestamp as to when it was updated. Git blame allows for use of multiple flags like many other git commands. Two flags you may find yourself using often are `-L` and `--reverse`.

With the -L flag, you can restrict your blame contents to a particular range of lines.
`git blame -L 2,4 README.md` will show you info on the last edit between lines 2 and 4
`git blame -L 2,+4 README.md` will show you info on the last edit starting at line 2 and includes the first 4 lines
`git blame -L 8,-4 README.md` will show you info on the last edit for the four lines preceding and including the 8th line.

Git blame is great for determining the who and when of an addition or modification to a line in a file, but what about a deletion? When you need to find the commit when a particular line of code was deleted, you can use `git blame --reverse <fromhash>..<tohash> <file>`. You'll need to find a commit when the line did exist in the file and use this as the from hash value. Git blame will start at this hash and list out all lines that existed in this file between the two hashes and specify the commit where they were last modified, including a deletion.

You may notice as you work with code that has been around a while, that the blame for a particular line does not correspond to the time when the line was added. The project may have gone through a refactor and had formatting "fixed" as some point. For these cases, the person listed in the blame may not be able to answer the questions you have about the intent of the code. In these cases, you can specify an exact commit hash to inspect with `git blame <hash> <filename>`.

Comparing changes

Git needs a mechanism by which it can create that change log. It also exposes that mechanism so that we can view every change that has ever been committed to our repositories with `git diff`. Without specifying any files, the diff will show you all changes that have been made to the repo since the last commit on the current branch. If you'd like to restrict that to specific files, you can specify them in the command: `git diff -- README.md`. You can also check all of the changes that have occurred between two commit hashes: `git diff <fromhash>..<tohash> -- <filename>`.

Working with multiple Remote Repositories

The forking workflow described above relies on having multiple remote repositories, but that may not be the only reason we might need to configure multiple repositories in our project. As an example, let's say that we run across an open source project with a lenient license such as the MIT license. This project would be ideal for an internal use case but would involve a bit of customization. Since the project is actively maintained, we would want to make sure that we could continue to benefit from improvements, but since our changes would be specific to our use case, it wouldn't make sense to request those changes be made back in the source project.

Accomplishing our goal will be similar to the forking workflow method but since we won't be request changes to be merged back to the project, we really don't need to maintain our fork on the same platform as the original project.

First, we start by simply cloning the project. Once the project is cloned, we can run `git remote -v` to list the current remotes and names. You'll see that the remote named "origin" is currently set to the main project's git URL. We are going to want to change that by giving the original project a different remote name. We can do that with `git remote rename origin upstream`. Now we need to create a new repository where our changes can live. Back in whatever git provider we use, we create a new repository just like we did at the beginning of the session. We want to make certain that when we create our new repository, that we do not initialize it with any files so that our new repository is empty. Once the new repository is create, we grab it's URL and set it to the URL of a new "origin" remote with `git remote add origin <URL>`.

Finally, we push up the entire contents up to our new remote by specifying the remote by name and a branch where the code should go. `git push -u origin master`. Now we can work on this project as it if were our own.

At some point, the maintainer of the original project may push some updates, either features or bug fixes, that we will also want to incorporate into our project. We do this the exact same was as we have done before with merging branches only this time we also specify the remote the change is coming from. `git fetch upstream master` then `git checkout master && git merge upstream/master`. Notice in the commands that the fetch command includes a space between the name of the remote and the branch whereas the merge command separates them by a "/".

Another user case for multiple remotes is if our team does use the forking method and we need to checkout a work-in-progress branch of one of our coworkers. In this case we can simply use the `git remote add <remotename> <URL of coworkers fork>` and then checkout their branch with `git fetch <remotename>` and `git checkout <remotename>/<branchname>`. Our coworker will need to have set up the permissions on their fork in such a way that would allow us to read and if they've set up permissions to allow us to push, we can even push changes to their fork by specifying their remote.

Common gotchas

When using Git, there are time when the anticipated behavior does not match up with the actual behavior. This is a non-exhaustive list of possible scenarios and fixes.

Filename case change

On certain operating systems (Mac included) the renaming of a file and changing only the case of the title is not implicitly including in the git change log. As an example, let's say you have a file called File_One.txt and for whatever reason you need to rename it to be all lower case. If you simply rename the file to file_one.txt, that new name will not be included in the change log. This will cause issues in the places where that file is referenced for anyone else using your repo. Everything will work on your system because the file name is properly named, but for everyone else, the file name will continue to have capital letters. Git includes its own method to rename files. In order for the name change to apply to everyone, you would need to use `git mv File_One.txt file_one.txt`. This will create a change in the log that can be staged and committed along with any other code change.

Files that share name of branches, tags, commits, etc

Git commands reference many things by name and sometimes the commands are multipurpose causing some issues. One example is the `git checkout` command. The checkout command not only lets you switch to a particular branch, but it also lets you reset any changes to a particular file to the state it was at the last commit for the branch you are on.

Let's say you have a file called "README.md" in your repo, you've made some changes to that file but want to revert all of those changes to the previous state. You can accomplish this with `git checkout README.md`. Now let's say that you've done the same thing to a file named "master". `git checkout master` isn't going to reset the file to the original content, it's going to switch your current branch to the one called "master". Git allows you to explicitly state that the name you are referencing is a file by placing it after two dashes. `git checkout -- master` will reset the file called "master" rather than checking out the master branch.

Key Terminology

- Repository
    - All of the files whose changes are being tracked
- Commit
    - A collection of individual changes to files in the repository
- Stage
    - Changes that are being prepared to be included in a commit
- Branch
    - A series of commits based on a single starting point
- GitHub Enterprise
    - The software that runs https://git.target.com

Resources

- Collection of Git topic tutorials from the people behind Jira, SourceTree and BitBucket
    - https://www.atlassian.com/git/tutorials
- Git GUI tools
    - SourceTree
    - GitHub Desktop
    - …more
- Target WIKI page for Git
    - https://wiki.target.com/tgtwiki/index.php/Git
- The "how to fix my git whoopsie" site
    - https://ohshitgit.com/