Reliant Git 101

Intro:

Reliant uses Git and Gitlab (a git repository hosting provider) for managing our code base (including Puppet code, RPM, portal, and other websites); therefore it's important to understand how to use this tool properly.

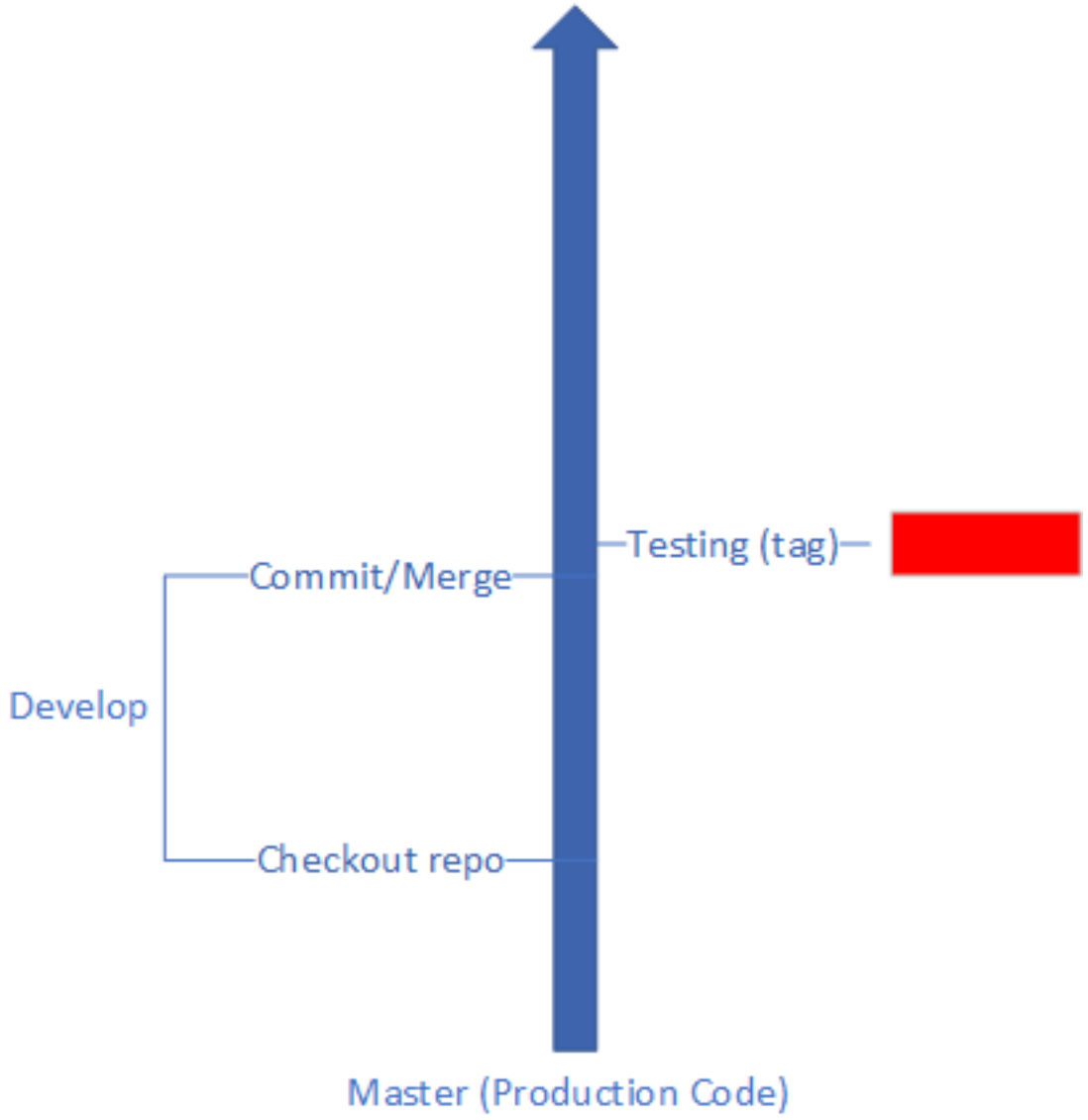
What is GIT?

Git is a command line, distributed version control system that works exceptionally well with text files and works ok with binary files (such as image files). Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It can be used locally to manage version control for files, as well as push updated code to remote repositories (repos), or pull from remote repos. This keeps you from having files like ddns, ddns-101518, ddns-101518b, and ddns-100318c.

Obviously the ddns-100318c is an older version, but what about the others? Git avoids this confusion. It also allows you to maintain a single folder structure for your code project, while allowing for different "branches", e.g. dev, prod, bugfix, etc. The alternative is to have almost duplicate code inside of sub-folders labelled "dev", "production", "bugfixes", etc.

Lastly, Git allows you to share your updated, **WORKING** code to the remote repos, allowing your team to do code reviews, make changes, update other sections of the code, revert back to previous working sections and even figure out who did what (in terms of commit messages, and blame).

Do NOT push broken code to a remote repo. Other team members may assume that it's working and attempt to update another section of code using your broken code as a dependency, thus breaking their code. This will waste a lot of time. If you can't get your code to work, then either don't commit your code at the end of the day or stash it. *(See below for git stash.)*



The Important Components

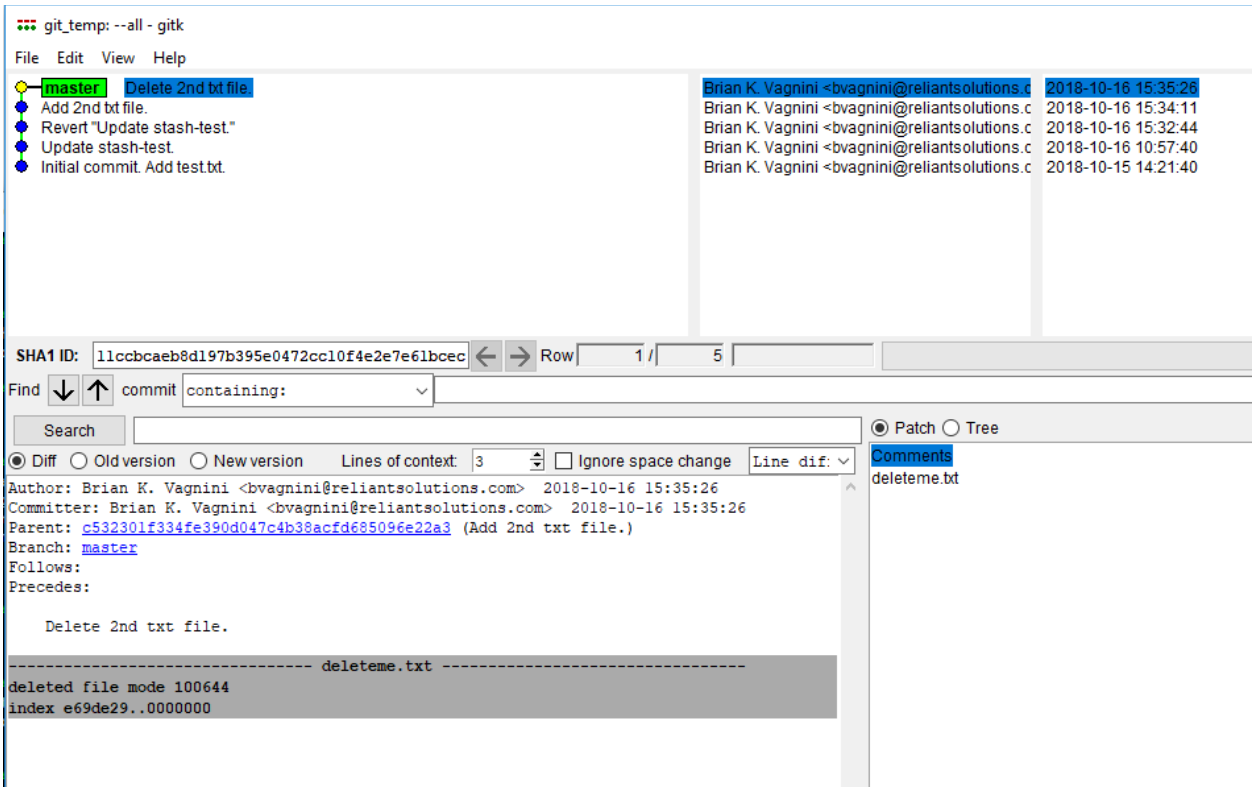
Component	Description
Git	This is the local install of git.
Git repos	These are the code bases that you want to be able to manage.
GitLab/git.reliant.io	This is the repo hosting provide that Reliant uses to manage and share code among ourselves. Repos that Reliant stores here are private, but some repos can be made public, meaning that people can view the source code anonymously.

SHA-1 checksum	This is a 40 character checksum that is used to identify each snapshot of your code (when you add things to the repo and commit them), and is unique. This means it's impossible to change the contents of any file or directory without Git knowing about it.
gitk --all	This is a visual tool that shows all the changes made to the repo, as well as all the branches and when / how they get merged back into the master (aka production) code base.
.gitignore file	This file tells git to ignore certain file types (such as .DS_Store (if using MacOSx), or a certain folder (e.g. data/ - useful is the data is temporary and used for processing).

git config --global	Use this to configure your email address and name - used by the commit messages. The --global makes this config change effective across ALL repos that you are working with.
	
	`git config --global user.email "inewperson@reliantsolutions.com"`
	`git config --global user.name "Ima Newperson"`

tag or release	A release is a tag that was created by our [http://puppetsvn/puppet/trunk/bin/release?view=markup] script.

gitk --all



How to use git?

There are three stages to git:

- Modified
- Staged
- Committed

Stage	Description
Modified	is the files that have changed but nothing has been done to add them to the repo for tracking purposes.
Staged	is when you add the files to the repo, but before you commit the files. Git is now aware that there are changes to be made, but they don't actually happen until you commit. Files don't stay in this stage very long.
Committed	is when you add the files to the repo, along with a commit message. This step creates the SHA-1 checksum. For the commit message, be brief, but descriptive in what the changes encompass. This is so you can look at the commit message and have some idea of the changes that took place.

There are three sections to a git project/repo:

- Working directory
- Staging area
- git repo

Section	Description
Working directory	is the local folder from where you are working.
Staging area	is a hidden folder in the .git folder (even in Windows) that contains (temporarily) the changes you've made but not committed.
git repo	is the latest commit that you've made. It is also in the .git folder (although it's not called "git repo" or even "repo". It's simply the last commit, and will be named with the SHA-1 checksum.)

A Look at the Logs

Logging works differently in git than it does with other applications. Logging shows the commit messages, and is called via "git log".

Usage of git log

Command	What it does
git log	Shows all logged commit messages, starting in reverse chronological order.
git log -5	Displays the last five commit messages.
git log --after= <date>	Show commits more recent than a specific date.

Basic Usage

Command	What it does	Notes

git init	initializes the repo	Do this command in your local machine terminal at the root of the project
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp`
		`$ pwd`
		`/c/Users/brian/projects/code/git_temp`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp`
		`$ git init`
		`Initialized empty Git repository in C:/Users/brian/projects/code/git_temp/.git/`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ ls -a`
		`./  ../  .git/`

git add .	adds ALL files at this level and below into the Staging Area of the repo	You can also add a specific file if someone else needs the updated code. This would be via "git add <filename>".
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ touch test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ git add .`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ git status`
		`On branch master`
		
		`No commits yet`
		
		`Changes to be committed:`
		`  (use "git rm --cached <file>..." to unstage)`
		
		`        new file:   test.txt`

git commit -m "Initial commit. Add test.txt."	This is what changes were made to the repo. Be descriptive, but brief.	This step will create the SHA-1 checksum and make the change to the local repo.
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ git commit -m "Initial commit. Add test.txt."`
		`[master (root-commit) be7fdc1] Initial commit. Add test.txt.`
		` 1 file changed, 0 insertions(+), 0 deletions(-)`
		` create mode 100644 test.txt`

git status	This command should be run first. It shows you the status of the repo.	`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)`
		`$ git status`
		`On branch master`
		`nothing to commit, working tree clean`

The typical LOCAL repo workflow is as follows:

1. git status
2. make changes to your local repo
3. git add .
4. git commit -m "<status message here>"
5. git status

How often do you commit?

- Whenever you make a major change. (Verify with Charles / Mike / Jason on this)

To work as a part of a team:

- There are a few additional things to consider.

Command	What it does	Notes

git remote -v	Shows you the status of any remote repos from which you can push or pull code	The remote repo has to be manually set up in advance, unless you cloned the repo in order to get it to your local machine. IN this case, the remote repos are already configured. (see below for cloning)
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/portal_git_repo/reliant-portal (v1.0.1)`
		`$ git remote -v`
		`origin  git@git.reliant.io:reliant/reliant-portal.git (fetch)`
		`origin  git@git.reliant.io:reliant/reliant-portal.git (push)`

git clone https://github.com/vinta/awesome-python.git	Creates a local copy of the repo wherever you are at in your terminal session	It will create the folder for the project for you e.g.(if you are cloning foo, you only need to be at c:\projects, NOT c:\projects\foo- the latter will wind up creating c:\projects\foo\foo
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code`
		`$ git clone https://github.com/vinta/awesome-python.git`
		`Cloning into 'awesome-python'...`
		`remote: Enumerating objects: 3653, done.`
		`remote: Total 3653 (delta 0), reused 0 (delta 0), pack-reused 3653`
		`Receiving objects: 100% (3653/3653), 3.61 MiB | 3.22 MiB/s, done.`
		`Resolving deltas: 100% (1820/1820), done.`

git stash
 	
or
 	
git stash save <message>
	Stash the changes in a dirty working directory away.	Use `git stash` when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the `HEAD` commit.
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ ls -a ./ ../ .git/ test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ touch stash-test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git status On branch master Untracked files: (use "git add <file>..." to include in what will be committed) stash-test.txt nothing added to commit but untracked files present (use "git add" to track)`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git stash No local changes to save // need to add it to staging first`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git status On branch master Untracked files: (use "git add <file>..." to include in what will be committed) stash-test.txt nothing added to commit but untracked files present (use "git add" to track)`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git add .`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git status On branch master`

		Changes to be committed:
		  (use "git reset HEAD <file>..." to unstage)
		
		  new file: stash-test.txt
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git stash
		Saved working directory and index state WIP on
		master: be7fdc1 Initial commit. A dd
		test.txt.
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git status
		On branch master
		nothing to commit, working tree clean
		//Note that there's no uncommitted changes
		present in the repo
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ ls -a
		./ ../ .git/ test.txt
		//Also note that the stash-test.txt file has
		apparently disappeared. This is because git
		reset the repo to the last committed version
		that had a clean working directory.

git stash list	Lists the stashes that you have	Use this command when you finally get back to what you were working on. brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git stash list stash@{0}: WIP on master: be7fdc1 Initial commit. Add test.txt.
git stash show	Shows the files that have been stashed	brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git stash show stash-test.txt | 0 1 file changed, 0 insertions(+), 0 deletions(-)

git stash apply	Reapplies the work in progress to your directory	Your working directory is now dirty again (meaning that you have saved work that's NOT been committed to the repo.). You will need to continue to fix your code to a WORKING status, then re-add and commit it to the repo.
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ ls -a ./  ../  .git/  test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git stash apply On branch master Changes to be committed:   (use "git reset HEAD <file>..." to unstage)`
		
		`        new file:   stash-test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ ls -a ./  ../  .git/  stash-test.txt  test.txt //YAY! Our work is back!`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git status On branch master Changes to be committed:   (use "git reset HEAD <file>..." to unstage)`
		
		`        new file:   stash-test.txt`
		
		Make your changes to the file that you were interrupted on.
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ vi stash-test.txt`
		
		`brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master) $ git status On branch master Changes to be committed:   (use "git reset HEAD <file>..." to unstage)`
		
		`        new file:   stash-test.txt`
		
		`Changes not staged for commit:   (use "git add <file>..." to update what will be committed)`

```
  (use "git checkout -- <file>..." to discard
changes in working directory)

        modified:   stash-test.txt


brian@LAPTOP-59N4GOFJ MINGW64
~/projects/code/git_temp (master)
$ git add .
warning: LF will be replaced by CRLF in stash-
test.txt.
The file will have its original line endings in
your working directory.

brian@LAPTOP-59N4GOFJ MINGW64
~/projects/code/git_temp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   stash-test.txt


brian@LAPTOP-59N4GOFJ MINGW64
~/projects/code/git_temp (master)
$ git commit -m "Update stash-test."
[master 393fa60] Update stash-test.
 1 file changed, 8 insertions(+)
 create mode 100644 stash-test.txt

brian@LAPTOP-59N4GOFJ MINGW64
~/projects/code/git_temp (master)
$ git status
On branch master
nothing to commit, working tree clean

brian@LAPTOP-59N4GOFJ MINGW64
~/projects/code/git_temp (master)
$ git stash list
stash@{0}: WIP on master: be7fdc1 Initial
commit. Add test.txt.
```

So how we get rid of the stash? I don't need it anymore, as I made the changes and fixed my code, and committed to the repo.

git stash clear	Gets rid of stashes.	brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git stash list
		stash@{0}: WIP on master: be7fdc1 Initial commit. Add test.txt.
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git stash clear
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ ls -a
		./ ../ .git/ stash-test.txt test.txt
		//Our file now exists since we committed it properly.
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git stash list
		//Our stash is gone.
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/git_temp (master)
		$ git status
		On branch master
		nothing to commit, working tree clean

git push origin master	This takes your local master branch and places it in the remote master branch at GitLab	Do this step once you have working code, at the end of your shift/day. brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/python/learning/p4e (master) $ git push GH-origin master Counting objects: 3, done. Delta compression using up to 4 threads. Compressing objects: 100% (2/2), done. Writing objects: 100% (3/3), 499 bytes | 499.00 KiB/s, done. Total 3 (delta 0), reused 0 (delta 0) remote: remote: Create a pull request for 'master' on GitHub by visiting: remote: https://github.com/bkvagnini/p4e/pull/new/master remote: To github.com:bkvagnini/p4e.git * [new branch] master -> master

git pull origin master	This takes your remote master branch at GitLab and places it in the local master branch	Do this step prior to beginning work on new code/code changes the following day. This pulls the code that's on the remote repo down to your machine, so that you are in sync with the remote repo.
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/python/learning/p4e (master)
		$ ls -a
		./ ../ .git/ lilbro.py
		//Only one file in the repo
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/python/learning/p4e (master)
		$ git pull GH-origin master
		remote: Enumerating objects: 4, done.
		remote: Counting objects: 100% (4/4), done.
		remote: Compressing objects: 100% (3/3), done.
		remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
		Unpacking objects: 100% (3/3), done.
		From github.com:bkvagnini/p4e
		* branch master -> FETCH_HEAD
		  301331e..d1e75e1 master -> GH-origin/master
		Updating 301331e..d1e75e1
		Fast-forward
		test-of-pull.txt | 2 ++
		1 file changed, 2 insertions(+)
		create mode 100644 test-of-pull.txt
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/python/learning/p4e (master)
		$ ls -a
		./ ../ .git/ lilbro.py test-of-pull.txt
		//After pull from the remote repo, there are now TWO files
		
		brian@LAPTOP-59N4GOFJ MINGW64
		~/projects/code/python/learning/p4e (master)
		$ git status
		On branch master
		nothing to commit, working tree clean
		//The pull request updated your local repo, so there's no adding or //committing that needs to be done on account of the new file

Branches

A branch is used when you are having to modify the existing code base, but still need to maintain the existing code base. A good example of this would be you have the production branch, which still needs to be left intact. Meanwhile, you are starting to work on the next version of it. The production branch will still require code changes (for bug fixes), but not for future features. Branches are the solution to this.

When you switch to another branch, the files will change, to include removing files or adding different files. Don't freak out if you don't see the files that you were expecting to see; you may simply be in the wrong branch.

Branch Naming Convention

<type>-<description>

```
feature-openvpn5
feature-sensu
feature-network-routing
issue-gitlab-78
issue-jira-276
```

Command	What it does
git branch	Displays a list of the available branches for the local copy of the repo
git checkout -b <branch>	Create a branch and change to it
git push origin <branchname>	Push your development branch to git.reliant.io (GitLab)
git branch -m new-name	**Rename your local branch.** If you are on the branch you want to rename
git branch -m old-name new-name	**Rename your local branch.** If you are on a different branch:
git push origin :old-name new-name	Delete the old-name **remote** branch and push the new-name local branch.

git rebase	Designed to integrate changes from one branch into another branch.
	
	`git checkout feature`
	`git rebase master`
	
	This moves the entire `feature` branch to begin on the tip of the `master` branch, effectively incorporating all of the new commits in `master`.
	
	But, instead of using a merge commit, rebasing *re-writes* the project history by creating brand new commits for each commit in the original branch.

Rebase

Assume the following history exists and the current branch is "topic":

```
          A---B---C topic
         /
    D---E---F---G master
```

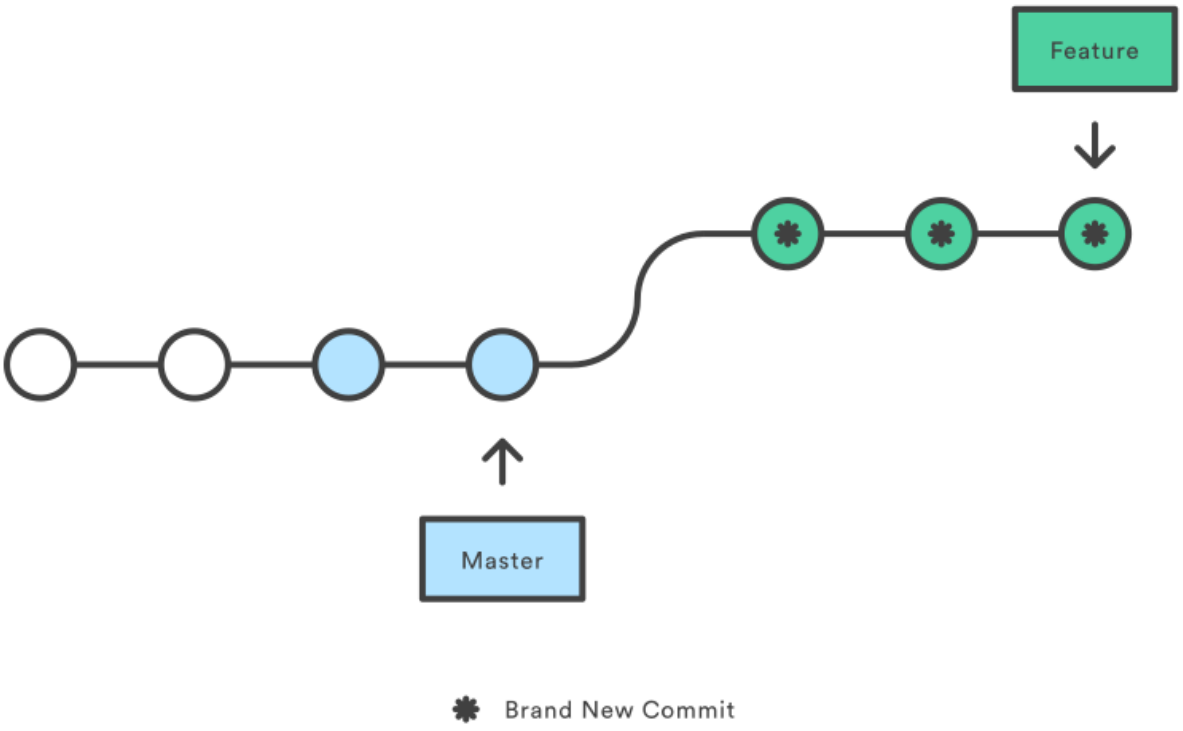
From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
                A'--B'--C' topic
               /
    D---E---F---G master
```

NOTE: The latter form is just a short-hand of `git checkout topic` followed by `git rebase master`. When rebase exits `topic` will remain the checked-out branch.



The typical Team repo workflow is as follows:

1. Ensure that you are on the correct branch (This step is VITAL)
2. git pull origin master

3. git checkout dev
4. git rebase master
5. git status
6. make changes to your local repo (dev branch)
7. git add .
8. git commit -m "<status message here>"
9. git status
10. Merge your changes to your local Dev branch with your local Master branch
11. Push the changes to the remote repo with git push origin master
12. Submit a merge request on GitLab

Scenario - I screwed up and committed non-working code. How do I "undo"?

Click here to expand...

git revert will undo a commit

```
git revert HEAD
or
git revert <SHA-1 number (first 6 digits is sufficient)>
```

Scenario - I want to change to another branch / delete my local dev branch

Click here to expand...

git branch <branch name> will create a branch.

git checkout <branch name> will switch to that branch.

```
git branch newdev
git checkout newdev

OR
git checkout -b newdev  (creates a new branch and switches to it)
brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch
* master

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch newdev
//creates a new branch


brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch
```

```
* master
  newdev
//shows that the new branch is available, but you are still on the master
branch

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git checkout newdev
Switched to branch 'newdev'

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
```

git branch -d<branch name> will delete a branch in your local repo once you are finished with it.

```
git branch -d newdev
brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch
* master
  newdev

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch -d newdev
Deleted branch newdev (was 11ccbca).

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git branch
* master
```

Scenario - I want to remove files in my branch

Click here to expand...

rm <file name> will remove files from the repo and notify git that it's happened. You MUST commit the change prior to switching branches; otherwise, the file will disappear from your master branch as well.

```
brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ ls -a
./  ../  .git/  deleteme.txt  test.txt
// deleteme.txt exists in newdev branch

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ rm deleteme.txt
// deleteme.txt is removed from newdev branch
```

```
brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ git status
On branch newdev
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    deleteme.txt

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

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ git add .

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ git commit -m "Delete 2nd txt file."
[newdev 11ccbca] Delete 2nd txt file.
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 deleteme.txt

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ ls -a
./  ../  .git/  test.txt
// deleteme.txt no longer exists in newdev branch

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ git checkout master
Switched to branch 'master'

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ ls -a
./  ../  .git/  deleteme.txt  test.txt


        deleted:    stash-test.txt

// deleteme.txt still exists in master branch
```

Scenario - I want to merge my local dev branch to my local master branch

Click here to expand...

From the branch that you want to merge your changes into (in this case, master) , do a git merge <source branch> <destination branch>.

We want to merge from newdev to master.

```
brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
$ ls -a
./  ../  .git/  test.txt
//deleteme.txt doesn't exist in newdev branch but does in master branch

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (newdev)
```

```
$ git checkout master
Switched to branch 'master'
//getting ready to merge

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ ls -a
./  ../  .git/  deleteme.txt  test.txt
//deleteme.txt still exists in master branch prior to the merge

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git merge newdev master
Updating c532301..11ccbca
Fast-forward
 deleteme.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 deleteme.txt

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ ls -a
./  ../  .git/  test.txt
//deleteme.txt no longer exists in master branch due to the merge

brian@LAPTOP-59N4GOFJ MINGW64 ~/projects/code/git_temp (master)
$ git status
On branch master
nothing to commit, working tree clean

//There are no changes to be committed to the master repo, as the merge
command did all of that behind the scenes
```

At this point, you can switch back to your dev branch and continue working.

Scenario - I want to make a release or a tag

Click here to expand...

Given that you create a branch / tag called "feature/dns-forwarding"

```
git checkout -b feature/dns-forwarding

//==== Adding files and committing ====
//If you add new files to the repository, you must remember to add them to
the repository.
//As an example:
git add somefile.pp
bin/commit -m "Descriptive commit message" -s rde

//In order to test the code you are developing, you'll need to run a
//puppetmaster instance at the customer site using the branch that you are
//developing on. For example, assuming you want to test the
//''feature-dns-forwarding'' branch in RDE:
```

```
 ssh -A login.rde
 ssh -A puppet
 /usr/local/sbin/puppet_slave_webrick.sh -b feature-dns-forwarding
//the -b is for build

//then find a lab box that you can use
//and run the following to test your puppet code
sudo puppet agent -t --masterport 8141

//port 8141 is the webrick puppet port- this will time out eventually
//anyone that is running puppet in their environment will be running against
port 8140, so
//there's no issue of screwing up a box in the field
```

Access needed

You will need to be granted read/write access to the specific repos on Gitlab prior to cloning the repos, and making changes to the code.

Pre-Req Module(s):

-
  - Reliant Puppet 4 - 101

Recommended Next Modules:

  - Puppet Modules 201
  - Puppet 4 Releases 101

Resources:

- https://en.wikipedia.org/wiki/Git
- https://git-scm.com/ (getting Git- This tool is cross platform.)
- https://about.gitlab.com/ (Gitlab)
- git.reliant.io (our gitlab account - MUST have at least READ access to a repo to sign in)
- https://github.com/ (to create your own acct, for side coding projects)
- Reliant Library
- https://github.s3.amazonaws.com/media/progit.epub
- https://www.git-scm.com/docs/git-checkout
- https://multiplestates.wordpress.com/2015/02/05/rename-a-local-and-remote-branch-in-git/
- https://www.atlassian.com/git/tutorials/merging-vs-rebasing
- https://git-scm.com/docs/git-stash
- https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow