# Git Training Documentation

***Release v0.9.4-6-g2971c09***

**Theodore A. Roth**

**Dec 05, 2017**

# CONTENTS:

CHAPTER

ONE

# INTRODUCTION

Git should be treated as a tool that makes your life as a developer easier and allows you to try new ideas without risking the integrity of your code base. Use of **`git`**, or any other source control tool, should not be treated as unnecessary paperwork or an afterthought. Use **`git`** for everything you do such that it becomes second nature. The mastery of **`git`** is a skill that all developers should attain.

This training is meant to be a hands on, exercise based introduction to **`git`**. **`git`** is a source code management tool that is widely used in the IT industry.

This document will *not* cover the following:

- Using **`git`** on Windows (although command line **`git`** via bash works).
- Using **`git`** from a GUI tool or from an IDE as I believe mastery of **`git`** on the command line is far more powerful and portable than what most GUI tools can provide (the may be some GUI tools out that are quite powerful, I don't use GUI tools, other than **`gitk`**, so I probably speak from ignorance here).

## 1.1 Objectives

- Create an empty local **`git`** repo.
- Make changes within your local repo.
- Working with branches in your local repo.
- Clone a remote repo.
- Publish your changes to a remote repo.
- Fetch/Merge/Pull changes from a remote repo.
- Collaborate with other developers.
- Select and negate commits using **`git-revert`** and **`git-cherry-pick`**.

## 1.2 The Pro Git Book

Much of the information for this document came directly from the Pro Git Book.

Reading this book from cover to cover and understanding it all will put you well on the way to becoming an expert git user.

Before going any further with this document, you are strongly encouraged to read the following chapters of the Pro Git Book:

- Chapter 1: Getting Started

- Chapter 2: Git Basics
- Chapter 3: Git Branching
- Appendix C: Git Commands

If you do that, you may not need this document, but it is hoped that this document will be a good companion to the book.

Also recommended reading after you have mastered the basics:

- Chapter 5: Distributed Git
- Chapter 7: Git Tools (some of these topics are touched in this document)

Reference links to the Pro Git Book are provided when relevant throughout this document.

## 1.3 Contributing to the Document

This is an open source document with the source project hosted on gitlab

> https://gitlab.com/OpenAVR/git-training

Feel free to use the issue tracker there to report problems or to request improvements to the document. Please include the version string, shown at the top and bottom of each page of the html, in the issue so that we can better determine how to address the issue.

If you want to help improve the document, please fork it and send merge requests.

## 1.4 Viewing the Document

This automatically generated html is viewable:

> http://openavr.gitlab.io/git-training/

A PDF version of the document is available for download:

> http://openavr.gitlab.io/git-training/GitTraining.pdf

These are automatically updated when changes are pushed to the master branch of the source project.

## 1.5 LICENSE

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.


## 1.6 Prerequisites

There are a few things you need to do to prepare for the training.

- Install git tools on your system.
  - **git**
  - **gitk**
  - a diff tool (**kdiff3**, **meld**, **xxdiff**, etc)
  - others???
- Create a user account on gitlab, github, or bitbucket (or any git hosting service you want to use; your company might host a service internally that you can use to collaborate with your co-workers).

CHAPTER

TWO

# SSH SETUP

In order to access a git service using git, you will need to create an ssh key pair and install it on the git service.

The following examples will use gitlab as an example. Other services may be slightly different in some respects, but the overall theory should be similar.

## 2.1 Generating a SSH key

If you have not created a ssh key, use the following command to generate one (do this from your workstation so you will have the key available for use after the training):

```
$ cd ~/.ssh
$ ssh-keygen -t ed25519 -f id_ed25519_git
```

Some git hosting services may not yet support `ed25519` keys. If the service you want to use does not support `ed25519` keys, generate your key pair with the following:

```
$ ssh-keygen -t rsa -b 4096 -f id_rsa_git
```

---

**Note:** ECC (Eliptic Curve Cryptography) keys are much smaller and less computationally intensive than RSA keys, while being equally or more secure. For RSA keys to become more secure, you need to increase the number of bits which increases both the size and the computational load to use the key. Also, it appears that the future of cryptographic keys will likely be ECC. With that in mind, you should probably start using `ed25519` keys when ever possible.

Reference:

https://wiki.archlinux.org/index.php/SSH_keys

---

Next, will you need to upload the public key (`id_ed25519_git.pub`) to the git repository hosting service.

---

**Note:** It is not required to name your key `id_ed25519_git`. Doing so only makes it clear to you what the key is used for. Any ssh key that you have already created can be used if you do not want to create another key. The only requirement is that a public key identifying you is installed on the hosting service's server.

---

---

**Note:** There is nothing stopping you from using the same ssh key for every server you need access to, but best practices dictate that you should generate a different key for each server.

---

**References:**

- https://git-scm.com/book/en/v2/Git-on-the-Server-Generating-Your-SSH-Public-Key

### 2.1.1 Upload Public Key to gitlab

Login to gitlab and install the contents of the public key file using this page:

https://gitlab.com/profile/keys

Test the key installation with (you should get your user name in the response instead of mine):

```
$ ssh -i ~/.ssh/id_ed25519_git git@gitlab.com
Enter passphrase for key '/home/troth/.ssh/id_ed25519_git':
PTY allocation request failed on channel 0
Welcome to GitLab, Theodore A. Roth!
Connection to gitlab.com closed.
```

### 2.1.2 Upload Public Key to github

Login to github and install the contents of the public key file using this page:

https://github.com/settings/keys

Test the key installation with:

```
$ ssh -i ~/.ssh/id_ed25519_git git@github.com
Enter passphrase for key '/home/troth/.ssh/id_ed25519_git':
PTY allocation request failed on channel 0
Hi troth! You've successfully authenticated, but GitHub does not provide shell access.
Connection to github.com closed.
```

### 2.1.3 Upload Public Key to bitbucket

Login to bitbucket and install the contents of the public key file using this page:

https://bitbucket.org/account/user/your-user-id/ssh-keys/

Test the key installation with:

```
$ ssh -i ~/.ssh/id_ed25519_git git@bitbucket.org
Enter passphrase for key '/home/troth/.ssh/id_ed25519_git':
PTY allocation request failed on channel 0
logged in as taroth.

You can use git or hg to connect to Bitbucket. Shell access is disabled.
Connection to bitbucket.org closed.
```

## 2.2 Adding an SSH Config Entry

You can make your life a little easier by adding an entry for the hosting service to your `~/.ssh/config` file.

Add the following to your `~/.ssh/config` file on your workstation:

```
$ cat >>~/.ssh/config <<EOF
Host gitlab
     User git
     Hostname gitlab.com
     Port 22
     Identityfile ~/.ssh/id_ed25519_git
EOF
```

Add similar entries for github or bitbucket should you wish to use those services.

With this configuration entry in place, you can now just use the following to test access to the service:

```
$ ssh gitlab
Enter passphrase for key '/home/troth/.ssh/keys/id_ed25519_git':
PTY allocation request failed on channel 0
Welcome to GitLab, Theodore A. Roth!
Connection to gitlab.com closed.
```

## 2.3 SSH Agent

When pushing and pulling from a remote **git** repository, **git** will use **ssh** to both authenticate you to the server and to encrypt the data passed between your system and the server.

Each time a **git** command communicates with the remote server, you will be asked to provide the pass phrase for your **ssh** key. This can get annoying and tedious quite quickly. There are two ways to avoid having to type a pass phrase every time:

- Generate an **ssh** key with no pass phrase. This is simple to do but is *bad practice*. If someone obtains your private key they can immediately use it to access your **git** repositories on the server (e.g. they could insert malicious code, delete branches, or any number of bad operations). Even worse, if you use the same key for many servers, they would have access to all of them. It is *not* recommended to use **ssh** keys with no pass phrase.
- Use **ssh-agent**. This tool will unlock a key by having you enter the pass phrase for the key once and continue to use that unlocked key for the duration of the session, or until the **ssh-agent** process is terminated.

On many systems, a **ssh-agent** is automatically started when you login to the desktop. You can check if a **ssh-agent** is running with the following:

```
$ env | grep SSH_AGENT
SSH_AGENT_PID=25422
```

If `SSH_AGENT_PID` is set, then **ssh-agent** is likely running. You can verify with the following:

```
$ ps ax | grep 'ssh-agen[t]'
25422 ?        Ss     0:00 ssh-agent
26904 ?        Ss     0:00 ssh-agent
```

Notice that on my system, I have two instances of **ssh-agent** running and one matches the `SSH_AGENT_PID` value. There is nothing wrong with running multiple instances of **ssh-agent**. One could have been started automatically when you logged in on the desktop and another could have been started by you via an **ssh** login from a remote system.

If you need to start the **ssh-agent**, the following is the easiest way (assuming you are running in a `bash` shell):

```
$ eval $(ssh-agent)
Agent pid 26904
```

which is equivalent to this:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-zkl1xk7JkO7W/agent.26903; export SSH_AUTH_SOCK;
SSH_AGENT_PID=26904; export SSH_AGENT_PID;
echo Agent pid 26904;
$ SSH_AUTH_SOCK=/tmp/ssh-zkl1xk7JkO7W/agent.26903; export SSH_AUTH_SOCK;
$ SSH_AGENT_PID=26904; export SSH_AGENT_PID;
$ echo Agent pid 26904;
Agent pid 26904
```

**Note:** Starting **ssh-agent** from the command line will make that instance of **ssh-agent** accessible only from within that shell session, not from other shells you have started from your desktop environment. You could export the variables as done above in other shells to share the **ssh-agent** with other shells currently running.

Once you have an instance of **ssh-agent** running, you will need to add keys to the agent:

```
$ ssh-add ~/.ssh/id_ed25519_gitlab
Enter passphrase for /home/troth/.ssh/id_ed25519_gitlab:
Identity added: /home/troth/.ssh/id_ed25519_gitlab (troth@example)
```

You can view which keys have been added to the agent with:

```
$ ssh-add -l
256 SHA256:WDXiVEhLF7Lt053cqvTXpxb62u+4Uv9e/sYnd9bEYno troth@example (ED25519)
```

Now any **git** commands (accessing the remote server via **ssh**) run from this shell or session will no longer need to have you enter the pass phrase to unlock the **ssh** key.

Of course, the **ssh-agent** is useful beyond the realm of **git**. Use it for any **ssh** or **scp** operations.

CHAPTER

THREE

# FIRST STEPS

There are a few book keeping things you should do before using git.

## 3.1 Set Your Identity

Git needs to know your name and email address which are used in commit messages. Use the following commands to set up your identity:

```
[1]$ git config --global user.name "John Doe"
[2]$ git config --global user.email johndoe@hp.com
```

These commands will modify your `~/.gitconfig` file which is your global config information.

You can override global config values in each repo by editing the `<repo>/.git/config` file.

**References:**

- https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

## 3.2 Getting Help

All git commands are well documented via git's online documentation:

```
[3]$ git help <verb>        # Full text of help (shows the man page version)
[4]$ git <verb> --help      # Usually equivalent to 'git help <verb>'
[5]$ git <verb> -h          # Usually less verbose, just listing options
[6]$ man git-<verb>
```

Try the following to get more information about git configuration:

```
[7]$ git help config
```

**References:**

- https://git-scm.com/book/en/v2/Getting-Started-Getting-Help

CHAPTER

# FOUR

# YOUR FIRST GIT REPO

**References:**

- https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository
- https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
- https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Before you can do anything useful with git, you will need a repository to work with. Let's create an empty directory which will become your first repository:

```
[1]$ cd ~
[2]$ mkdir -p repos/example
[3]$ cd repos/example
```

Creating a repo with git is as simple as this:

```
[4]$ git init
```

To see what your empty repo looks like:

```
[5]$ find .
```

Create a new file in your repo:

```
[6]$ cat > hello <<EOF
#!/bin/bash
echo "Hello world!"
EOF
[7]$ chmod 755 hello
[8]$ ./hello
```

Get a status of the repo:

```
[9]$ git status
```

Tell git you want to track the file:

```
[10]$ git add hello
[11]$ git status
```

And finally, commit the staged file to the repo:

```
[12]$ git commit
[13]$ git log
[14]$ git status
```

Let's make some changes to `hello`:

```
[15]$ echo 'echo "goodbye"' >> hello
[16]$ git status
[17]$ git diff
[18]$ git add hello
[19]$ git status
[20]$ git diff
[21]$ git diff --cached
```

We now have some staged changes. Let's make another change before we commit:

```
[22]$ echo 'exit 0' >> hello
[23]$ git status
[24]$ git diff
[25]$ git diff --cached
```

At this point, we can do one of two things:

- Commit the staged changes, then add the unstaged changes and commit again. This will give us two separate commits:

  ```
  [26]$ git commit
  [27]$ git add hello
  [28]$ git diff --cached
  [29]$ git commit
  [30]$ git log
  ```

- Stage the second change and commit. This will combine the two changes into a single change which will be a single commit:

  ```
  [31]$ git add hello
  [32]$ git diff --cached
  [33]$ git commit
  [34]$ git log
  ```

Try both of these methods (make a different change the seconds time around) and compare the results.

At this point, you should have a basic understanding of the **`git add`**, **`git status`**, **`git diff`** and **`git commit`** commands.

CHAPTER

FIVE

# BRANCHING

**References:**

- https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

Working with branches in your local repo is so easy with git that using feature branches is the normal workflow for most users of git.

Unlike with subversion, branches and tags are distinctly different. A branch is dynamic and moves with each commit. Whereas a tag is static and points to a specific point in the history of the repository.

A default `master` branch is usually created by **`git`** during repository initialization. It is best to think about this branch as equivalent to Subversions `trunk`. In theory, you can rename or delete the `master` branch, but it's best to just leave it alone.

## 5.1 Creating Branches

Let's create a branch off of `master` in our `~/repos/example/` repository we created early:

```
[1]$ cd ~/repos/example
[2]$ git checkout master
[3]$ git branch
[4]$ git branch train/stuff
[5]$ git branch
[6]$ git checkout train/stuff
[7]$ git branch
```

Now you should have a branch called `train/stuff` and have it checked out.

You can also checkout and create a branch at the same time:

```
[8]$ git checkout -b train/doodle master
[9]$ git branch
```

Notice that we were still on the `train/stuff` branch, but we created a new branch called `train/doodle` relative to `master` without having to checkout `master` first.

**References:**

- https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

## 5.2 Merging Branches

Let's make a change on the `train/doodle` branch:

```
[10]$ echo '# Change on doodle branch' >> hello
[11]$ git add hello
[12]$ git commit
```

Now make a change on the `train/stuff` branch:

```
[13]$ git checkout train/stuff
[14]$ echo '# Change on stuff branch' >> hello
[15]$ git add hello
[16]$ git commit
```

Now the fun begins. Let's go back to the `master` branch and merge the changes from the `train/doodle` and `train/stuff` branches:

```
[17]$ git checkout master
[18]$ git merge train/doodle
[19]$ git merge train/stuff    # <-- Oops! We got a merge conflict!
```

Before we can move on with our work, we need to fix the merge conflict:

```
[20]$ git mergetool
[21]$ git status
[22]$ git diff
[23]$ git diff --cached
[24]$ git commit
[25]$ git log --oneline --graph
```

**Note:** You can configure git to use the mergetool of your choice (see *Diff Tool and Merge Tool Configuration*).

## 5.3 Deleting Branches

Once you have merged all the changes on your feature branch back to `master`, you can delete the branch:

```
[26]$ git checkout master
[27]$ git branch -d train/doodle
[28]$ git branch
```

You can also delete a branch that you consider a dead end without merging it back to `master` (or some other branch):

```
[29]$ git checkout -b train/deadend master
[30]$ echo '# this is a deadend branch' >> hello
[31]$ git add hello
[32]$ git commit
[33]$ git checkout master
[34]$ git branch
[35]$ git branch -d train/deadend  # <-- Notice that git fails here
[36]$ git branch -D train/deadend
[37]$ git branch
```

**References:**

- https://git-scm.com/book/en/v2/Git-Branching-Branch-Management

CHAPTER

SIX

# STASHING WORK IN PROGRESS

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning

Now that you are comfortable with branching, we will explore the **`git stash`** command.

Create a new branch for a new feature:

```
[1]$ cd ~/repos/example
[2]$ git checkout -b feature/foo master
[3]$ git branch
```

Make another change to the `hello` file:

```
[4]$ echo '# Adding the "foo" feature' >> hello
[5]$ git status
[6]$ git diff
```

Time goes by and you have made a bunch of changes, but you have not added your changes and you are not ready to commit your work in progress.

Then your boss comes by and tells you to work on Jira ticket AA-1234. Obviously, you don't want to work on the ticket in the `feature/foo` branch or on the `master` branch, but on its own branch. Let's try switching to a new branch for the ticket:

```
[7]$ git checkout -b feature/AA-1234 master
[8]$ git status
[9]$ git diff
```

Notice that our pending change got carried over to the new branch. That's not want we wanted. Switch back to the `feature/foo` branch:

```
[10]$ git checkout feature/foo
```

What we need to do is *stash* away the changes on the `feature/foo` branch:

```
[11]$ git stash save 'wip: new foo feature'
[12]$ git stash list
[13]$ git stash show -p
```

Since the changes to `feature/foo` are safely stashed away, we can switch to the `feature/AA-1234` branch and work on that change:

```
[14]$ git checkout feature/AA-1234
[15]$ git status
```

```
[16]$ sed -e '/^exit/s/exit 0/exit 1/' hello > hello.tmp
[17]$ mv hello.tmp hello
[18]$ git status
[19]$ git diff
```

At this point, you get blocked on the Jira ticket, and want to jump back to the `feature/foo`. Just stash the work in progress on the `feature/AA-1234` branch:

```
[20]$ git stash save 'wip: fix for AA-1234'
```

You should now see two stashes:

```
[21]$ git stash list
stash@{0}: On feature/AA-1234: wip: fix for AA-1234
stash@{1}: On feature/foo: wip: new foo feature
```

Switch back to the `feature/foo` and put the stashed changes back in place:

```
[22]$ git checkout feature/foo
[23]$ git stash pop stash@{1}
[24]$ git stash list
[25]$ git status
[26]$ git diff
```

Commit the changes for `feature/foo`:

```
[27]$ git add hello
[28]$ git diff --cached
[29]$ git commit
```

Now that we're done with adding the `feature/foo` branch, we can go back to the `feature/AA-1234` branch and commit the stashed changes there:

```
[30]$ git checkout feature/AA-1234
[31]$ git stash list
[32]$ git stash pop
[33]$ git diff
[34]$ git add hello
[35]$ git commit
```

Both the `feature/foo` branch and the `feature/AA-1234` branch are ready to be merged into the `master` branch.

---

**Note:** Students should now merge both branches into master as an exercise. You should expect merge conflicts.

---

**Warning:** It is not a good idea to leave things in the stash for very long. It's better to push to a WIP branch on a remote server (more on that later). Best practice is to only stash changes for a minimal amount of time and then keep the stash empty when the stashed changes are no longer needed. It's not fun to find something in your stash that is months old and you can not remember what it was.

For more information:

```
$ git help stash
```

CHAPTER

SEVEN

# REMOTE REPOSITORIES

**References:**

- https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Gits distributed nature makes collaboration extremely easy. It is very easy to have your local repository associated with another repo either on the local system or on a remote system. You can even associate your repo with any number of other repos.

Before you can access the repo hosting server (gitlab, github, bitbucket, etc) using **`git`**, be sure you have created an account and setup your ssh key (see *SSH Setup*).

For the sake of simplicity, gitlab will be used as the hosting service in the examples and exercises that follow.

## 7.1 Create the `git training playground` Repository

To facilitate the exercises which follow, a `git training playground` repository needs to be created. If you want to work collaboratively with others, multiple people will need access this repo.

For convenience, a template `git training playground` has been created:

https://gitlab.com/OpenAVR/git-training-playground

Fork this repository by clicking on the `Fork` button. This fork will be your personal remote repo. You will push your changes to this fork and share it with other collaborators.

---

**Note:** Forking repository using the web UI on the cloud hosting services will create clone of the parent repository in your account (or into a group or team in which you have permission to create repos) on the server. This new repo is linked back to the parent repo such that when you push changes into your fork, you will be able to generate merge requests back to the parent repo (merge requests will be covered later). You can also make a fork of a fork and generate merge requests from your fork back to the intermediate fork.

---

---

**Note:** If you do not wish to use the template `git training playground`, you can create a new empty repository on gitlab for your use (note that it will not have the files referenced below). Log in to gitlab and create the repo using the **`New Project`** menu item which should land you on this page:

https://gitlab.com/projects/new

---

## 7.2 Cloning A Remote Repo

We're now going to create a local copy of a remote repo using the **`git clone`** command. The rest of this training session will involve working with the `git-training-play.git` repository which resides on the gitlab server (replace `<userid>` in what follows with the your user id on gitlab).

This `git-training-play.git` is the `git training playground` repo that was referred to in the *Prerequisites*.

Clone the `git-training-play.git` repo:

```
[1]$ cd ~/repos
[2]$ git clone gitlab:<userid>/git-training-play.git playground
[3]$ cd testing
```

**Note:** I'm using `gitlab:` as a short hand (defined in my `~/.ssh/config`) which is equivalent to `git@gitlab:`

From now on, we will assume that we are working in the `~/repos/playground/` directory.

To see what remote repos you have configured, run the following command:

```
[4]$ git remote -v
```

## 7.3 Pushing to Remote Repos

Let's create a feature branch where we will do our collaboration work (replace `<name>` with your name or user id, but keep it simple):

```
[5]$ cd ~/repos/playground
[6]$ git branch -a
[7]$ git checkout -b feature/train-<name> origin/master
[8]$ git branch
```

Add your name to the students list:

```
[9]$ vim students.py
```

Commit your change to the local branch:

```
[10]$ git add students.py
[11]$ git commit
```

Now push your changes up to gitlab:

```
[12]$ git push origin feature/train-<name>
```

Verify that your changes are on the gitlab server via the web interface.

Using the gitlab web interface, generate a merge request and assign it to one of your collaborators (or yourself if not working with a group).

## 7.4 Pulling from a Collaborators Repository

One great feature of **git** is the ability to have one local repository which can pull from any number of remote repositories. When you first clone a repo (like we did above), there will be a single remote called `origin`. Use the following to see the remotes configured for your local repo:

```
[13]$ git remote -v
```

At this point, you will need to find a collaborator, teammate or buddy that is working through these exercises. You can setup your local repository to pull in changes that they have pushed up to their remote on the gitlab server.

Adding your buddy's repo on gitlab as one of your remotes, is done with the following command:

```
[14]$ git remote add <buddy> gitlab:<buddy>/git-training-play.git
```

Now pull in their feature branch:

```
[15]$ git pull <buddy> feature/train-<buddy>
```

You should now have their changes in your branch:

```
[16]$ git log
```

Once you have pulled from your partner and resolved any merge conflicts, push your local repo out to your gitlab repo:

```
[17]$ git push origin feature/train-<name>
```

Once your buddy has pushed to his gitlab repo, pull his changes again. You should now both be in sync.

## 7.5 Using a Shared Team Repository

For the next exercise, we will all make changes to our local repo and push them directly into the `gitlab:<team>/git-training-play.git` repo and bypass your personal `gitlab:<user>/git-training-play.git` repo.

Someone on your team (or group of buddies) will need to fork the template `git training playground` and add all of the team members as members of the forked project. Each team member will need to be assigned a role which allows them to push to the `git training playground` repo. On gitlab, the roles the members will need to be `developer`, `master` or `owner`.

Each member of the team will need to add the team repo as remote:

```
[18]$ git remote add <team> gitlab:<team>/git-training-play.git
```

Create a branch for the team to collaborate on, and push it to the server (only one person on the team needs to do this):

```
[19]$ git checkout -b feature/train-<team> master
[20]$ git push team feature/train-<team>
```

Each team member will need to fetch all of the branches from the team remote and checkout a local version of the branch:

```
[20]$ git fetch <team>
[21]$ git branch -a                    # show all branches for all remotes
[22]$ git checkout -b feature/train-<team> <team>/feature/train-<team>
```

Make some changes to the `students.py` file and publish your changes to the remote:

```
[23]$ vim students.py
[24]$ git add students.py
[25]$ git commit
[26]$ git pull team feature/train-<team>
[27]$ git mergetool                          # resolve conflicts if needed.
[28]$ git push team feature/train-<team>
```

You should keep pushing and pulling from the origin repo until you have pulled everyone else's changes without getting a merge conflict.

## 7.6 Merge and Pull Requests

After you have committed changes locally to your clone of a project on which you are working, you will likely want to get your changes into the upstream git repository. Most of the time you will not have permissions to push directly into a git repository you forked (or the central shared repo where you work has restrictions on who can push into certain branches, e.g. `master`).

The work flow in the situation where you need to work through a fork typically follows this pattern:

1. Fork the parent repo into your account (via gitlab Web UI).
2. Clone forked repo from your account, not the upstream parent repo.
3. Make changes locally.
4. Commit locally to a feature branch.
5. Push your feature branch into your remote fork on gitlab.
6. Use the gitlab Web UI to create the merge request from your fork.
7. The maintainer of the parent repo will either merge your changes into `master`, make comments for you to address and resubmit or reject the request outright.

Replace gitlab in the above with the hosting service you are using.

The work flow when you can push a feature branch to a shared, centralized parent repo, but are not allowed to push directly into the `master` branch follows this pattern:

1. Clone upstream repo.
2. Make changes locally.
3. Commit locally to a feature branch.
4. Push you commits into a feature branch on the centralized upstream repo.
5. Create a merge request from your branch on the `master` branch.
6. Another person on the team will need to review the request and either merge, comment or reject the request.

See the Pro Git Book for detailed discussions on contributing to a project:

> https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project

### 7.6.1 Merge/Pull Requests on Hosting Services

On gitlab, *Merge Requests* are how you notify the parent of your fork that you would like them to merge code you have pushed into your fork into their repository:

https://gitlab.com/help/user/project/merge_requests/index.md

*Merge Requests* on github are called *Pull Requests* (but will work the same as *Merge Requests* on gitlab):

https://help.github.com/articles/about-pull-requests/

Similarly on bitbucket:

https://confluence.atlassian.com/bitbucket/tutorial-learn-about-pull-requests-in-bitbucket-cloud-774243385.html

### 7.6.2 Pull Requests via Command Line

You can also generate a *Pull Request* from the **git** from the command line with the **git request-pull**.

This is used to send and email request to an upstream maintainer to pull your changes from you public repository.

You will probably not use this method when working with the cloud hosting services, although it is how Linux Kernel maintainers submit changes to the maintainer up the hierarchy of maintainers.

For more information:

```
$ git help request-pull
```

## 7.7 Deleting a Branch from a Remote Repo

Sometimes you will need to delete a branch from a remote repo you own. Let's push a dummy branch to gitlab:

```
[29]$ git checkout master
[30]$ git push origin master:tmp/dummy
```

On second thought, you decide that was not a good idea. You can delete the branch on the remote quite easily:

```
[31]$ git push origin :tmp/dummy
```

This probably looks a little odd. From the **git help push** command:

> The format of a <refspec> parameter is an optional plus +,
> followed by the source ref <src>, followed by a colon :, followed
> by the destination ref <dst>. It is used to specify with what
> <src> object the <dst> ref in the remote repository is to be
> updated.
>
> ...
>
> Pushing an empty <src> allows you to delete the <dst> ref from the
> remote repository.

## 7.8 Deleting a Repo from a Hosting Service

It is possible to delete an entire repo from the gitlab server.

This varies from service to service. It can be done, but use caution when deciding to delete a repository from the hosting service.

The how-to is left as an exercise for the student. Recommend creating a private repo on the hosting service and digging into the repo settings to figure out how to delete it.

CHAPTER

EIGHT

# GIT AND SVN

It is possible to use **git** as a front end to subversion repositories using git-svn(1).

The **git-svn** is also very useful when you need to convert an SVN repository into a git repository with little to no loss of history.

## 8.1 Creating a Git Repo from an SVN Repo

**Todo:** Add discussion of **git-svn**

See: https://git-scm.com/book/en/v2/Git-and-Other-Systems-Git-as-a-Client

CHAPTER

NINE

# GIT IGNORE FILES

**References:**

- https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#_ignoring

Git has support for ignoring files via its `.gitignore` files.

This is extremely useful since it is usually desirable to avoid checking in files that are generated during a build. The `gitignore` files will also reduce the noise when you run **`git status`**.

Git's `.gitignore` files are similar to CVS's `.cvsignore` files and Subversion's **`svn:ignore`** property. One major difference between them is a `.gitignore` will affect all subdirectories whereas CVS and Subversion only ignore files in the current directory. For example, if you put the following in the `.gitignore` file at the top level of your project:

```
*.o
*.bak
tmp
```

then any file or directory within any directory of the repository that matches those patterns will be ignored by git.

Git will also exclude files which match patterns defined in `<repo>/.git/info/exclude`.

For more detailed information on `.gitignore` files:

```
$ git help ignore
```

CHAPTER

# TEN

# ALTERING HISTORY

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

Git provides some tools and commands which allow you to alter the history of your repository. Sometimes it's good to be able to change history, but it can be dangerous.

It all boils down to one simple rule:

> **Warning:** Once you have shared your changes with another repository (by either you pushing, or someone else pulling), those changes should not be changed. Doing so will cause all kinds of grief for the user of the other repository. There are cases where pushing altered history is appropriate, but you need to make sure it is the right thing to do.

**Note:** Modern version of **`git`** are much more forgiving when you pull changes from a remote branch which has altered its history. Still, make sure you understand the consequences of pushing altered history.

The following are some commands that change history. This discussion is only provided to make you aware of the existence of these features, not to be a comprehensive overview. If you need to use these commands, you should do some more research into them to gain a better understanding of the pros and cons of the commands.

## 10.1 `git rebase`

The **`git rebase`** command is used to alter where a sequence of commits is based.

For example, you created a branch off master, then you make a series of changes. While you were making your changes, master has evolved. Rebasing your branch to the current master will make all of your changes relative to the last change on master.

By comparison, doing a merge of master onto your branch would place all of the changes since you branched on top of your changes.

## 10.2 `git rebase -i`

Interactive rebasing allows you to change alter the history on a commit by commit basis. Some of the things you can do with interactive rebasing include:

- Change the ordering of commits.

- Alter commit messages.
- Modify a commit.

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History#_changing_multiple

## 10.3 `git commit --amend`

The **`git commit --amend`** command allows you to squash your current change into the last committed change.

For example, you just committed a change and then you realize that you introduced a typo. Using **`git commit --amend`** allows you to fix the typo and the history will never even show that the typo existed.

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History#_git_amend

## 10.4 `git reset`

The **`git reset`** command adjusts the HEAD ref to a given commit.

For example, you just committed a change and you realize that you really didn't want to make that change. You could revert it, but that would leave history of the bad change. Using **`git reset`** you can rewind history so that HEAD points to the previous commit (or any prior commit you want).

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

## 10.5 `git filter-branch`

Rewriting history on the entire repository can be done with **`git filter-branch`**. This is a really big hammer that can be extremely powerful, while also extremely dangerous.

> **Warning:** Be very careful with **`git filter-branch`**. Recommend creating a new clone before you start trying to use this.

Typical uses:

- Remove a file from all history (as if it never existed).
- Change an Author's email address in the entire history.

---

**Todo:** Discuss `git filter-branch`

---

> **Warning:** Think twice (or thrice) before using this tool. You will likely make enemies.

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History#_the_nuclear_option_filter_branch

CHAPTER

# ELEVEN

# TIPS AND TRICKS

Git can do a lot more things than we have seen during this training. This section will cover some of the more interesting things you can do with git.

## 11.1 git grep

The **git grep** command is useful for searching through all of the files that are under revision control. Files that are not under revision control are ignored.

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Searching

## 11.2 git add -i

The **git add -i** command makes the **git add** command interactive. You can select which files to add, update or patch. It is quite powerful and well worth learning how to use.

- https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging

I also find **git add -p** to be quite useful.

For more information:

```
$ git help add
```

## 11.3 gitk

The **gitk** tool provides a graphic display of the history of the repository. To use **gitk**, you may need to install the **gitk** package (using apt-get, yum or whatever your system uses to manage packages).

**gitk** shows a very dense view of the history of the repository by showing the following:

- Branches (local and remote).
- Which commit is currently checked out.
- Tags (and associated meta data).
- Commit relationships.
- Commit meta data.

To view the history for all branches at once:

```
$ gitk --all
```

Sometimes it is useful to pull down changes from a remote, but review them before merging locally:

```
$ git checkout master
$ git fetch origin
$ gitk origin/master
$ git merge origin/master
```

Another useful operation is to view the history of a specific file or directory:

```
$ gitk -- path/to/file
$ gitk -- path/to/directory
```

Right clicking on the subject line of a commit in **gitk** will bring a context menu that will allow you to before some basic **git** operations.

I often leave **gitk** running in the background as I work through changes and then use the `File->Reload` menu option to refresh the view of the repository.

Using **gitk** as part of this workflow can greatly improve the quality of your commits and make your fellow developers lives easier as you will be able to find and fix bad commits before you push them to other:

```
$ git checkout -b feature/cool-stuff master
$ vim <files>           # do some work
$ git status            # see what you changed
$ git diff              # review your work
$ git diff --check      # Check for white space errors in your changes
$ gitk                  # review you work WRT to history
$ git add <files>       # stage your changes
$ gitk                  # more review WRT to history
$ git ci                # commit locally
```

Iterate on the above to generate a series of commits and get ready to push:

```
$ gitk                  # Review the patchset you have generated locally
$ git rebase -i <ref>   # Cleanup your ugly or embarressing commits
$ gitk                  # More review (are you starting to see pattern developing?)
$ git push origin feature/cool-stuff    # publish your changes to the world for peer-
↪review
```

Now, you may be asking yourself, that seems like a lot of extra work? Yes, it is, but asking for peer-review of your code for things that you should have fixed before pushing is extra work for you and the reviewer later. And then if the review misses a bug that manifests a year later, how much extra work will it be to fix that bug then?

## 11.4 git diff --check

You can have **git** check for white space errors in your changes before you commit them with the **git diff --check** or **git diff --cached --check**. You should strive to use these commands before every commit.

Many Open Source project maintainers get grumpy if merge requests contain white space issues.

## 11.5 git describe

**Todo:** Discuss `git describe` and how it can be used to automate versioning.

## 11.6 Showing the Git Branch in your Prompt

If you are using **bash** as your shell, you can modify the command prompt using the `PS1` environment variable. If you install **git** and **bash_completion**, you will have a `/etc/bash_completion.d/git-prompt` file installed on your system. The `git-prompt` script should get sourced by your shell at login and will provide a **__git_ps1** shell function.

Using the **__git_ps1** shell function in your `PS1` variable will show the currently checked out branch in your prompt.

Here's what my prompt looks like (except for colorization):

```
### [11:35:29 am][Ubuntu-16.04] Branch: master
### [troth@example:~/git/git-train/git-training/source]
>--> $
```

And here's the code from my `~/.bash_profile` to generate that prompt:

```
OS_REL=$(lsb_release -irs | tr '\n' ' ' | tr ' ' '-' | sed -e 's/-$//')
export OS_REL

_BLK_ ()
{
    printf -- "\033[0m"
}
_RED_ ()
{
    printf -- "\033[31m"
}
_GRN_ ()
{
    printf -- "\033[32m"
}
_BLU_ ()
{
    printf -- "\033[34m"
}
_CYN_ ()
{
    printf -- "\033[36m"
}

PS1='\n$(_GRN_)### $(_BLU_)[\T \D{%P}]$(_CYN_)[${OS_REL}]'\
'$(__git_ps1 "$(_RED_) Branch: %s")\n'\
'$(_GRN_)### [\u@\h:\w]$(_BLK_)\n>--> \$ '

export PS1
```

Showing the currently checked out branch in the prompt saves you from constantly having to run `git branch` to see which branch you are working on.

## 11.7 Writing Good Commit Messages

Writing a good commit message is an art and takes practice, but is crucial to the sanity of future developers looking at your commit (your future self included).

The following is probably one of the best articles on writing **`git`** commit messages:

https://chris.beams.io/posts/git-commit/

## 11.8 Crafting a Good Commit

**Todo:** Discuss what makes a good commit and/or commit series.

CHAPTER

TWELVE

# ADVANCED GIT OPERATIONS

## 12.1 Tagging

**References:**

- https://git-scm.com/book/en/v2/Git-Basics-Tagging

**Todo:** Discuss tagging (annotated vs. non-annotated).

## 12.2 Cherry Picking Commits

**References:**

- https://git-scm.com/book/en/v2/Distributed-Git-Maintaining-a-Project#_rebase_cherry_pick
- https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Patching

**Todo:** Discuss cherry picking commits.

## 12.3 Reverting Commits

**References:**

- https://git-scm.com/book/en/v2/Git-Tools-Advanced-Merging#_reverse_commit
- https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Patching

**Todo:** Discuss reverting commits.

## 12.4 Aliases

You can create *aliases* for **`git`** commands that you use often to save you some typing.

I have added the following aliases to my `~/.gitconfig` file:

```
[alias]
    st = status
    br = branch
    co = checkout
    ci = commit -s
```

Aliases are optional. Use them if you find them useful.

More info:

```
$ git help config               # Search for 'alias'
```

**References:**

- https://git-scm.com/book/en/v2/Git-Basics-Git-Aliases

## 12.5 Generating and Applying Patches

**References:**

- https://git-scm.com/book/en/v2/Distributed-Git-Maintaining-a-Project#_patches_from_email
- https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project#_project_over_email
- https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Email

**Todo:** Discuss `git format-patch` and `git am`.

## 12.6 Diff Tool and Merge Tool Configuration

Using a diff tool or a merge tool is almost required when viewing complicated diffs or attempting to resolve a difficult merge conflict. Trying to resolve a difficult merge manually in your editor is madness and will likely result in an incorrect merge. Merge conflicts can be hard to resolve, so do yourself a favor and spent the time to find a merge tool that works for you and learn how to use it. You will be happy that you did.

Various diff tool and merge tool options are available:

- kdiff3
- vimdiff
- meld
- kompare
- xxdiff

There are many other options, but the above are the ones that I have tried.

Having tried all of the above, I have found **`kdiff3`** to work the best for me.

To configure **`git`** to use **`kdiff3`** when running **`git mergetool`** or **`git difftool`**, I added the following to my `~/.gitconfig` file:

```
[merge]
    tool = kdiff3
[mergetool]
    keepBackup = false
[mergetool "kdiff3"]
    cmd = /usr/bin/kdiff3 $BASE $LOCAL $REMOTE -o $MERGED
    trustExitCode = false
[diff]
    tool = kdiff3
    renames = copies
[difftool "kdiff3"]
    cmd = /usr/bin/kdiff3 $LOCAL $REMOTE
```

If you wish to use another tool, this may work with some tweaking.

## 12.7 Splitting Directories out of a Repository

**Todo:** Show how to split a repo in to multiple repos using **`git subtree split`**.

## 12.8 Working with multiple Repositories

More often than not, you will be working on a product that is built from multiple git repositories (Google Android is an extreme example of this). Fortunately, there are tools that can help make working with many repositories easier.

### 12.8.1 Google Repo

Google Repo makes cloning and managing a group of git repositories easier through the use of an XML manifest file. The manifest file lives in its own git repository which can be branched and tagged to represent a branching and tagging of the group as a whole.

> https://code.google.com/archive/p/git-repo/

If you need to write or maintain a manifest file, you'll need to read this:

> https://gerrit.googlesource.com/git-repo/+/master/docs/manifest-format.txt

Here's an example project which uses a Google Repo manifest file:

> https://github.com/openavr-org/imx7-base-manifest

### 12.8.2 Git Multi

The **`git-multi`** is an enhancement to **`git`** which allows you to run a **`git`** command on multiple repositories at the same time. Particularly useful for me are the following:

```
$ git multi status
$ git multi grep 'some-stuff'
```

My implementation of **`git-multi`** knows how to use Google Repo manifest files to determine which repos to operate on, although it can also work without Google Repo:

https://github.com/openavr-org/git-openavr

CHAPTER

# THIRTEEN

# ADDITIONAL RESOURCES

After finishing this training, you will likely be interested in some additional resources related to Git.

## 13.1 Git Web Site

The Git Web Site has loads of information:

http://git-scm.com/

## 13.2 Pro Git Book

The Pro Git book is freely available online:

https://git-scm.com/book

## 13.3 Git Repository Hosting Services

There are many cloud based services for git repositories. The following are the big three. There are many more features of each service than provided here, do your own research to see which suits your needs best.

### 13.3.1 gitlab

https://gitlab.com/

- Free public and private repositories.
- GitLab Pages can be used to host docs: https://about.gitlab.com/features/pages/

### 13.3.2 github

https://github.com/

- Free for public repositories.
- Must pay for private repositories.
- GitHub Pages are a great way to host websites.

### 13.3.3 bitbucket

https://bitbucket.org/

- Free public and private repositories.
- Need to pay if you need to give more than five people access to a private repo.
- Great Jira integration since Atlassian runs both services.

CHAPTER

# FOURTEEN

# TODO LIST

**Todo:** Discuss tagging (annotated vs. non-annotated).

(The original entry is located in /builds/OpenAVR/git-training/source/advanced.rst, line 37.)

**Todo:** Discuss cherry picking commits.

(The original entry is located in /builds/OpenAVR/git-training/source/advanced.rst, line 49.)

**Todo:** Discuss reverting commits.

(The original entry is located in /builds/OpenAVR/git-training/source/advanced.rst, line 61.)

**Todo:** Discuss `git format-patch` and `git am`.

(The original entry is located in /builds/OpenAVR/git-training/source/advanced.rst, line 102.)

**Todo:** Show how to split a repo in to multiple repos using **`git subtree split`**.

(The original entry is located in /builds/OpenAVR/git-training/source/advanced.rst, line 163.)

**Todo:** Discuss `git filter-branch`

(The original entry is located in /builds/OpenAVR/git-training/source/altering-history.rst, line 143.)

**Todo:** Add discussion of **`git-svn`**

(The original entry is located in /builds/OpenAVR/git-training/source/git-svn.rst, line 40.)

**Todo:** Discuss `git describe` and how it can be used to automate versioning.

(The original entry is located in /builds/OpenAVR/git-training/source/tips.rst, line 149.)

**Todo:** Discuss what makes a good commit and/or commit series.

(The original entry is located in /builds/OpenAVR/git-training/source/tips.rst, line 224.)

CHAPTER

FIFTEEN

# INDICES AND TABLES

- genindex

# INDEX