# DataGrokr Git Learning Module

Hi and welcome to the DataGrokr Git learning path! This moduile is designed to get you upskilled quickly on distributed version control using `Git`.

This learning path assumes no previous experience with version control, but basic familiarity with command line interfaces is required.

The learning module is designed to be self contained and has links to appropriate resources wherever relevant. This need not limit you just this document, feel free to consult other resources as well!

The following topics are covered in this learning module:

- What is Git? What separates it from alternatives?
- Internals of Git (index and objects)
- Managing source code & artifacts using Git
- Assignment to reinforce understanding

The learning resources are available in the `theory` directory, and the contents are as follows:

- Introduction to Git
  - What is distributed version control system?
  - What is Git and why was it created?
  - Why use Git?
  - Popular Git hosting providers
- Git Fundamentals
  - Repository
  - Branch
  - Commit
  - (OPTIONAL) Git internals
    - Index in Git
    - Git objects
- Repositories
  - Creating a Repository
  - Remote repositories
  - Fetching Branch Updates
- Branches
  - Working with Branches
  - Merging

  - Merge Conflicts
  - Rebasing
  - Viewing history
  - Branching strategies
  - (OPTIONAL) Further Reading
- Commits
  - Adding commits to a branch
  - Removing commits from a branch
  - Squashing commits

The assignment is spread across two mandatory parts:

- Part-1: Git Fundamentals:
  - Creating repositories
  - Branching
  - Adding changes via commits
  - Merging changes between branches
  - Basic history alteration
- Part-2: Git Hygiene
  - Writing a README
  - Using .gitignore
  - Raising clean pull requests
  - Review and approval best practices
  - Following team standards

At this point, you will already be familiarized with the basics of Git to get by on your day to day tasks, while also being exposed to the Do's and Dont's of collaborating using Git repo hosting sites (GitHub, BitBucket, GitLab, etc.).

There is also a completely optional, non-mandatory module that is available for folks looking for a tough challenge. This part comprises of a set of challenges that test the limits of your Git knowledge and if done, can aid you in the wild wild west of versioning!

- [TODO] Part-3: Wild Wild West
  - Destructive history rewrites
  - Cherry picking commits
  - Versioning LARGE files
  - Simple post receive hook

  **NOTE:** Throughout this learning module, we will be heavily referring the official Git documentation available in a very user friendly, free, textbook format at `https://git-scm.com/book/en/v2` (henceworth referred to as "the Git book"), rather than video tutorials or random internet tutorials. One of the key aspects of good developer is the ability to patiently consult official sources of information for software/libraries that you are using BEFORE Googling or checking on StackOverflow. This is a good chance to do just that!

### Setting up your environment

The Git book covers setting up `git` on *nix and Windows systems here.

### Errata

Please submit any questions, doubts, queries, issues or errors to either of these people:

- Dhruv (dhruv.chakraborty@datagrokr.com)
- Koushtav (koushtav.chakrabarty@datagrokr.com)
- Mustaf (mustaf.hussain@datagrokr.com)
- Rahul (rahul.patra@datagrokr.com)

## Introduction to Git

This submodule covers the basics of distributed version control using Git.

### What is distributed version control system?

A dstributed version control system (DVCS) is a way of doing version control where there is no central server. In DVCSs, in addition to the repository contents (source code, build artifacts, etc), a copy of the **repository** itself is stored. This copy includes all changes and entire history of the project. So the normal workflow involves someone creating a Git repo and hosting it on a site like GitHub or BitBucket and other members **clone** this repository. In their local systems, they now have the source code as well as all repository details. They make changes independently and in a **non-linear** manner and send changes to the **remote** repository to sync changes. Also, this **remote** copy of the repository on a Git hosting provider is the only central component.

There are ways to manage Git workflows not involving Git hosting providers. For example, you can have a system (on premise or on the cloud) which has a Git repository and developers clone this repository and then push changes back to this repository.

Either way, it is recommended to have this **remote** repository because things like adding authentication for collaborators becomes easy. In an enterprise, you don't want any random person to be able to contribute, or even look at the source code!

### What is Git and why was it created?

Git is a distributed version control system that was created by Linus Torvalds (yeah, the same guy that spawned the Linux project) to version control the source code of the Linux kernel.

It is now an independent open source project and is responsible for developing the `git` CLI and binary versions for popular OSs like *nix, Windows and Mac.

Some of the fundamental design philosophies of Git are:

- Applying patches should be as fast as possible
- Should be able to track hundreds and thousands of **objects** in the repository.
- Consistency and integrity of **objects**

As a DVCS, it allows a non-linear development mode and provides an exhaustive CLI interface for making and tracking changes in a source code repository.

Key features to remember about Git:

- It is a DVCS: Meaning it allows a distributed, non-linear development mode of collaboration among developers working on a common code base.
- Uses snapshots to store changes: Unlike older version control tools that tracks the **changes** that happen in between **revisions**, Git stores **snapshots** of the revisions in between **changes**. This is a fancy way of saying that if the contents of a file is updated, Git will store a copy of the file before the update, and then the updated version, but not the changes themselves. Changes or **differences** can be obtained by just taking two **snapshots** and **comparing** what changed
- Objects are content addressable: Git does not rely on filenames to uniquely identify objects, but rather uses the filename along with a SHA-1 hash of the contents of the files themselves (we will see what that means in the next submodule). This is the heart and soul of Git.

## Why use Git?

Version Control is an essential concept to know, understand and have a practical knowledge about if you are working as software engineer, no matter what subdomain you might be working in. Git offers one of the most simplest ways to manage different versions of your source code, and is one of the *de-facto* tools in the arsenal of a "rockstar developer*. Most enterprise projects now-a-days use either Git or Mercurial. And these enterprises have tens and hundreds of tools, libraries and software that are critical to sustaining the business. The ability to ensure that all effort spent in engineering a solution is stored and “lost” is critical and hence version control systems are essential.

## Popular Git hosting providers

These are some of the most popular Git hosting providers. All of them allow creating organizations, teams, private and public repositories. They also offer enterprise versions of their products with support and additional bells and whistles.

- GitHub (github.com)

- BitBucket (bitbucket.org)
- GitLab (gitlab.com)

Using Git hosting providers eases things like continuous integration and adding authentication (via HTTPS or SSH access).

# Git Fundamentals

This submodule covers various fundamental concepts of the internal workings of Git. The end of the submodule also directs you to the Git book for an even more in depth look at these concepts (not mandatory, but highly encouraged).

## Fundamental entities in Git

These are some of the fundamental "entities" that you will deal with when using Git:

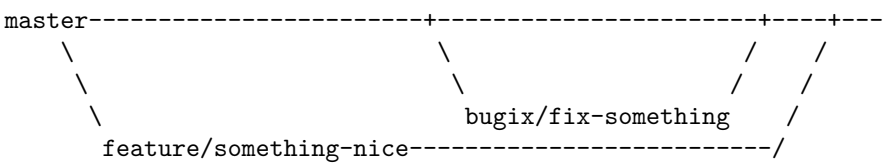
### Repository

A **repository** is a collection of the source code files and artifacts plus the **tracking info** that allows keeping track of changes.

If you create a new empty Git repository in a directory, a hidden directory called `.git` is created which stores this **tracking info**.

A repository has various **branches** that represent different states of the source code at any point of time. There is always at least one main development branch (usually called `master`) that contains the most stable version of the source code at any point in time. Several `feature` or `bugffix` branches may be created off of this main development branch to introduce new features or address bugs. Once development of a feature is complete or a bug is fixed, it is **merged** back to the main development branch.

This flow over time can be depicted with this illustration:

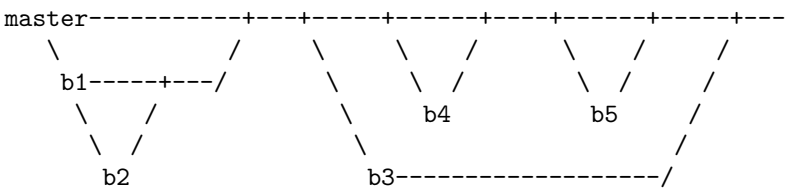



Here two people are working simulataneously on a feature and a bug by creating branches off of the main development branch (which is `master`), and then **merging** their changes to `master`, totally independent of each other.

### Branch

A **branch** is a way of creating a path of development without affecting other branches, and more importantly, the main development branch. The benefit of

branching is that multiple develoeprs can work on the same source code base, without stepping on each others' toes, and when they are done with their task, can simply merge their changes back to the main development branch.



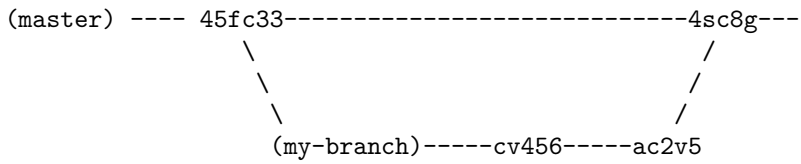

In this example, it can be seen that branch `b1` is created off of `master` and another branch `b2` is created off of `b1`. This illustrates the scenario where while working on a feature the developer encountered unknowns, and to experiment midway their changes, they created a new branch off of their feature branch and tried their experiments. When they were done, they merged the results of their experiments to the branch in which they were working on the feature. Another scenario also illustrated here with branches `b3`, `b4` and `b5`. Here, someone worked on a feature that took a considerably long time to develop, during which two other features were already added to the source code via `b4` and `b5`.

In essence, the main development **tree** consists of the main development branch, with other feature and bugfix branches being created off of it and then merged back to it.

### Commit

Commits can be considered as the **change set** that is already added to the history of the repository. A commit consists of the change that were added to the repository in some branch. When a source branch is merged to a destination branch, e.g., merging a feature branch to the main development branch, the commits in the feature branch are added to the main development branch. What this results in is the addition of the changes that were made in the feature branch to the main development branch.

Let's look at an illustration involving commits.




In this example, the main development branch `master` contains the commit `45c33` initially. Then someone created the branch `my-branch` off of `master` and added two commits `cv456` and `ac2v5` to `my-branch`. Once, `my-branch` was merged with `master`, `master` contains three new commits: `cv456`, `ac2v5` and `4sc8g` (this third one is an additional commit called the merge commit. This is

created by default and there are ways to disable this. This detail is helpful to know but does not concern us right now).

The funny looking string identifiers for the commits are the commit hashes.

### (OPTIONAL) Git Internals

> **Note**: Despite this section being optional, we highly encourage sticking through this and getting your hands dirty.

Though we can get by on our day to day tasks by knowing just the above fundamental details about Git, it is always helpful to know what and how commits, and branches work and how Git tracks changes.

Instead of coming up with our own story explaining this, we will refer you to the Git book which goes to extreme detail in illustrating the fundamental data structures that drive Git: index and objects.

The reference can be found here: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects.

## Repositories

This module explains how to create and work with Git repositories.

### Creating a Repository

Spin up your favourite terminal emulator and navigate to a suitable directory of your choice for these steps.

To create an new, empty Git repository, use the `git init` command:

```
$ git init
Initialized empty Git repository in /some/path/to/repository/.git/
```

This causes the creation of the directory `.git` in the directory where you ran the command. If you take a look at the contents of this directory, you should see something like this:

```
$ ls -1 .git/
branches
config
description
HEAD
hooks
info
objects
refs
$ ls -1 .git/refs/
```

`heads`
`tags`

Let us briefly discuss some of these subdirectories:

- `objects`: This directory contains details about all objects (text files and binary files) currently tracked by Git in this repository
- `refs/heads`: This directory contains the a pointer called `HEAD` to the latest commit in the current branch
- `refs/tags`: Commits have illegible hash values as identifiers. Human readble and friend names can be assigned to them and they are stored here.

More options available with this command can be checked here: `https://git-scm.com/docs/git-init`.

## Remote Repositories

In the real world, a Git hosting provider is used to facilitate development amongst the team, and the above step to create an empty repository and the main development branch is usually already taken care of at the start of the project.

What we are more interested is in how you can **clone** a **remote** repository and start contributing. There are two ways to clone a Git repository, via HTTP(s) or SSH, depending upon how the hosting provider has been configured to allow access to collaborators.

Cloning a repo via HTTPS:

```
$ git clone https://github.com/torvalds/linux.git
```

Cloning a repo via SSH:

```
$ git clone git@github.com:torvalds/linux.git
```

Cloning via SSH requires you to generate a SSH key, adding the public keys to the hosting provider for the repo that you are trying to access (either yourself if you have the rights to do that, else handing it over to someone who will do it for you).

More details for how to do that on GitHub can be found here. For other hosting providers like Bitbucket and GitLab, the key generation step remains the same, only difference is in the UI to add it.

Once you are able to clone remote repositories, you are ready to start branching, adding commits, etc.

## Fetching Branch Updates

Since multiple developers work on the same repository and create & delete branches all the time, it is essential to be able to **fetch** these updates to the

repository.

The command `git fetch` fetches the changes to the repository, which includes additions or deletions of branches:

Please consult this detailed guide to performing `git fetch` from this excellent document from Atlassian: https://www.atlassian.com/git/tutorials/syncing/git-fetch.

# Branches

This submodule covers branching commands and how typical branching patterns look like.

## Working with Branches

Let us use the repository of the highly popular `create-react-app` project to understand the basics of branching. The repository lives on GitHub at https://github.com/facebook/create-react-app.

Navigate to a directory of your choice and clone the repository, and switch to the `create-react-app` directory.

```
$ git clone https://github.com/facebook/create-react-app.git
Cloning into 'create-react-app'...
remote: Enumerating objects: 24382, done.
remote: Total 24382 (delta 0), reused 0 (delta 0), pack-reused 24382
Receiving objects: 100% (24382/24382), 14.17 MiB | 5.15 MiB/s, done.
Resolving deltas: 100% (15114/15114), done.
$ cd create-react-app
```

### Listing the active branch

To list the current branch you are on , you use the `git branch` command:

```
$ git branch
* master
```

### Listing all branches

To list all **local and remote** branches, you use the flag `-a` with the command:

```
$ git branch -a
* master
  remotes/origin/0.9.x
  remotes/origin/HEAD -> origin/master
  remotes/origin/add-babel-plugin-optimize-react
  remotes/origin/dependabot/npm_and_yarn/docusaurus/website/elliptic-6.5.3
  remotes/origin/dependabot/npm_and_yarn/docusaurus/website/lodash-4.17.19
```

```
remotes/origin/dependabot/npm_and_yarn/docusaurus/website/websocket-extensions-0.1.4
remotes/origin/feat/audits
remotes/origin/gh-pages
remotes/origin/github-actions-ci
remotes/origin/master
```

> **Note**: Since this a popular and fast paced project, the branches that you see might be different from what is shown here. Do not panic!

Some observations:

- The identifier `remotes/origin/` indicates that these branches are on the remote repository too.
- The branch `remotes/origin/HEAD` is a **special** branch that always indicates what is the current head on remote. In this case it is master.
- Though the `remotes/origin/` identifier refers to the asbolute branch names, you can instead refer to them instead by their identifiers just by the short names, i.e., `0.9.x` instead of `remotes/origin/0.9.x`

### Switching branches

To switch to an *existing* local branch, you use the `git checkout` command. Let us switch to the `add-babel-plugin-optimize-react` branch:

```
$ git checkout add-babel-plugin-optimize-react
Branch 'add-babel-plugin-optimize-react' set up to track remote branch 'add-babel-plugin-opt
Switched to a new branch 'add-babel-plugin-optimize-react'
```

If you run the `git branch` command now, you will see that Git now correctly identifies `add-babel-plugin-optimize-react` as the current active branch:

```
$ git branch
* add-babel-plugin-optimize-react
  master
```

Now, switch back to the master branch:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

### Creating a new branch

You can "branch off" from the **current active branch** using the `git checkout` command with the flag `-b` and specifying a branch name:

```
$ git checkout -b my-branch
Switched to a new branch 'my-branch'
```

Creating a new branch also causes it to become the active branch, which can be verified using `git branch`:

```
$ git branch
  add-babel-plugin-optimize-react
  master
* my-branch
```

### Deleting branches

Since Git is distributed, any branches that you create can not only be present on your local repository, but also on the remote repository. This happens when you update the remote repository to **push** your changes there.

To delete a branch **only** from your local repository, use the `git branch` command with the `-d` flag and specify the branch name to delete:

```
$ git branch -d my-branch
```

To delete a branch **only** from the remote repository, use the `git push origin` command with the the `--delete` flag and specify the branch name:

```
$ git push origin --delete my-branch
```

> **Note**: Before deleting a branch, you must switch to a different branch. The active branch cannot be the branch you are deleting.

### Updating local branches from remote

To **pull** changes made to branches on the remote repository, where other developers in your team check in their changes, to your local repository, we use the `git pull` command.

But this is done on a per branch basis. So, to pull the latest changes from remote for the `master` branch, we first checkout the `master` branch and then run:

```
$ git pull origin master
```

## Merging

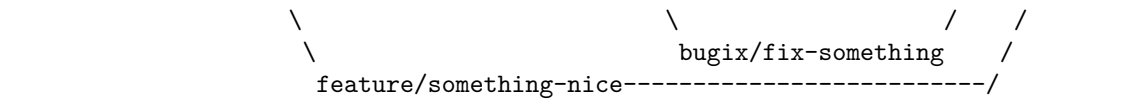
As was stated in the "Git Fundamentals" submodule, Git allows a non-linear workflow for development, meaning all changes can be applied independent of each other (as long as the changes don't have dependencies and block each other), simultaneously by multiple developers.

There is a **main development branch** which is from which branches are created and then changes added in these branches are "merged" back to.

In simple words, the act of merging is "adding" the changes (i.e., commits) you made one one branch, to a diferent branch.

To illustrate this, consider the following history tree:

```
master-----------------------+----------------------+----+---
    \                         \                    /    /
```




Here, there were two branches, `feature/something-nice` and `bugfix/fix-something` where created off of `master`, which is the main development branch, and once the changes required were implemented, merged again back to `master`.

The `git merge` command is used to merge a source branch to the current active branch.

An illustrative example (do not execute this):

```
$ git checkout master
$ git merge feature/something-nice
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Here, we are mergining the changes of branch `feature/something-nice` to `master`.

> **Note**: In the real world, we don't merge directly like this. We use **pull requests**, or popularly "PRs", to merge features or bugfixes to the main development branch. We will be taking a look at how to create a PR in the Part-2 of the assignment.

## Merge Conflicts

Even though Git facilitates developers working independently on different features/bugs simultaneously, it still does not prevent them from stepping on each others toes!

In the course of development, there will be cases that the same file is updated by two or more people. And a special case of this is when two or more people update the exact same places, i.e., lines, in the same files. Suppose, two developers updated line number 420 in the file `index.js` separately, and one of them merged their changes to the main development branch before the other. When the second developer tries to merge their changes to the main development branch, they will get an error, stating that the merge cannot proceed.

This is precisely what a merge conflict is: There exists two "versions" of "different" changes on the same line on the same file, and Git cannot automatically deduce which change to keep.

An illustration of this is shown below (do not execute):

```
$ git merge my-branch
Auto-merging index.js
CONFLICT (content): Merge conflict in index.js
```

```
Automatic merge failed; fix conflicts and then commit the result.
...
```

In case a merge conflict happens, it is upto the developer to understand why the conflict happen and decide "how" to resolve it. The "how" is very very situation dependent and unfortunately cannot be illustrated via a simple example, but is generally checking what changes happened prior to yours and then deciding whose change to keep: only the changes already merged before yours, or only your changes, or both.

If you use the `git status` command to check the state of the index in the current branch, you will see something like this (do not execute):

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

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

    both modified:      index.js

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

If you open the file `index.js`, you will see that **conflict markers** are placed in the lines a conflict has been detected:

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

The first block between the `<<<<<<` and `=======` markers is the change present in the **current** branch. The second block between the `=======` and `>>>>>>>` is what is being merged to the current branch.

The developer at this point will accept either or both of these changes and will then remove the conflict markers, then **commit** their changes, at which point the merge is complete and the conflicts are resolved.

## Rebasing

Merging is one of adding the changes in branch to another one. There is another alternative called **rebasing** that does the same, but works differently.

What `git merge` does is it **adds** the commits made on the source branch to the destination branch to which the changes of the source branch are being merged.

Rebasing, which is done using the command `git rebase`, on the other hand, takes the **patch** of changes that were made in the source branch, and then **replays** them in the destination branch. A patch is nothing but the **set of changes** that can be applied directly to destination branch to bring it to the exact same state as the source branch.

Here is an illustration (do not execute):

```
$ git checkout my-feature-branch
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
...
```

## Viewing history

You can view the history of the branch you are currently on using the `git log` command.

Coming back to the clone of the `create-react-app` repository we made earlier, if you run `git log` on any of the branches, you should see the commits made on that branch, which includes details like the commit author, date and time it was made, and a commit message (which is a brief description of the changes made in that commit).

```
commit 3d74b79de0ca40d3e89531207f8064a17207bfb8 (HEAD -> master, origin/master, origin/HEAD,
Author: Ian Sutherland <ian@iansutherland.ca>
Date:   Wed Aug 5 11:30:04 2020 -0600

    Prepare 4.0.0 alpha release

commit 50368252ed40b2838373c4007fc841ac17b950ef
Author: Ian Sutherland <ian@iansutherland.ca>
Date:   Wed Aug 5 11:25:10 2020 -0600

    Fix template name handling (#9412)

commit 6cd382621f5e3b9779e0576f2244196a8e976207
Author: Lenard Pratt <striderman34@gmail.com>
Date:   Wed Aug 5 18:23:48 2020 +0100

    Upgrade whatwg-fetch (#9392)

    Co-authored-by: Ian Schmitz <ianschmitz@gmail.com>
...
```

## Branching strategies

As observed multiple times by now, a typical repository has a main development branch, usually named as `master`, and feature and bugfix branches are created off of `master`. Once features or bugfixes are complete, they are merged back to `master`.

Since in enterprises, you rarely work single handedly on a project, it is important to know what branching strategy your team follows.

There are three popular branching strategies:

- Trunk based
- Gitflow
- Forking workflow

### Gitflow

In this Git workflow, we have a stable branch, usually `master`, a development branch, branched off of `master`, usually `develop`. Developers branch off of `develop` when working on new features (where the branch name is prefixed with `feature/*`), or addressing a bug (where the branch name is prefixed with `bugfix/*`). Fixes for time critical production bugs, known as hotfixes, are applied by branching off of `master` to create a branch prefixed with `hotfix/*`, and then merged directly to `master`. Then `master` is rebased to `develop`. Once the development branch, `develop`, has enough features for the next release cycle, we merge `develop` to `master`. Once the features are tested (QA automation, UAT, etc.), release branches are created off of master, prefixed with `release/*`.

Use Gitflow when the team size is large (more than 10 developers)

Detailed overview of Gitflow: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

### Trunk Based

This is more or less similar to Gitflow, except that it has less number of branches to worry about. Here, we have the `master` branch that serves as the main development branch and also the branch from which release candidates are created. Developers create feature, bugfix and hotfixes off of `master` and then merge their work back to `master`. Release candidates are created off of `master` by **tagging** specific commits.

Use trunk based when the team size is small (less than 10 developers).

Detailed overview of trunk based development: https://cloud.google.com/solutions/devops/devops-tech-trunk-based-development.

#### Forking Workflow

In this workflow, there is typically a single branch, which serves as both stable and development branches, similar to trunk based workflow. Every developer **forks** the repository, creating a new copy of the repository, and the apply changes directly on this branch, then they raise a pull request for `master` from their copy of the repository to the main repository.

Detailed overview of forking workflow: https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow.

### (OPTIONAL) Further Reading

You can consult the Git book for more interesting tid-bits about branching: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

## Commits

This submodule covers what commits are how to work with them.

### Adding commits to a branch

As discussed earlier, a commit is a sort of **snapshot** of the repository at a specific point of time, and contains the changes made in this snapshot, as well as details like the author, data & time, and a commit message (and optionall a description).

Commits are the basic building blocks of a repository. You create changes and then indicate to Git these changes by adding a commit. Merging and rebasing cause these changes to the main development branch. Each and every commit on a repository has a unique identifier, called the commit hash, or popularly, the commit "SHA" which is an SHA-1 hash value of the "changes" in the commit (it is not trivial to explain how this is generated here, please consult the Git internals optional section in the "Git Fundamentals" learning submodule).

The typical way to add commits is to create a feature or bugfix branch off of the main development branch, modify or delete existing files, add new files, and then commit them to the branch. This creates a snapshot of the repository with these new changes in this branch. Before we commit the changes, we need to **stage** them. The staging area is a sort of buffer that holds the changes that you want to save for commiting to the history of the repository, but "not yet" tracked in the history. Once you commit all staged changes, these are added to the history.

Illustration of adding a commit:

```
$ git branch
* master
```

```
$ git checkout -b feature/some-awesome-thing
Switched to a new branch 'feature/some-awesome-thing'
```

Then suppose you add a new file, `awesome.js` to the repo in this branch, containing an awesome feature.

To commit this file, we first need to **stage** it:

```
$ git add awesome.js
```

Once it is staged, it can be committed to the repository:

```
$ git commit -m 'Added an awesome feature'
```

If you were to now execute `git log` on this branch, you would see the last commit in the history.

## Removing commits from a branch

As with adding commits, Git allows removing commits from the HEAD of the branch too (we can delete commits from the middle too, but that is an advanced use case which we don't want to concern ourselves with at the moment).

Suppose, for whatever reason, you now want to undo the last commit (lets say it has a commit SHA a12g3). There are two ways to do this using the `git reset` command.

### Non-destructive way

If we do a **soft** reset, Git just reverts back the HEAD to *some* previous commit in the current branch. All changes applied as part of the reset are unstaged as well. But the important thing here is the changes are just unstaged, not discarded. Meaning, you still have the unstaged and uncommitted changes at your disposal.

Here's an illustration of removing the last commit:

```
$ git reset --soft HEAD~1
```

The numerical value after `HEAD~` indicates the number of commits from the top we want to revert back.

### The destructive way

Doing a **hard** reset causes Git to revert back the HEAD to *some* previous commit in the current branch. All changes applied as part of the reset are unstaged as well. But unlike a soft reset, a hard reset also **discards** these changes, so that your branch has not modifications at all.

Here's an illustration of removing the last commit via a hard reset:

```
$ git reset --hard HEAD~1
```

The numerical value after `HEAD~` indicates the number of commits from the top we want to revert back.

## Squashing commits

One of the important abilities of Git is that you can merge several commits, spanning several changes, into a single commit. There are several cases where you might need this. Let us take a look at one.

Suppose that you are woring on a complicated feature that needs a lot of experimentation and you make these changes in small increments. Once you are done adding the feature, you notice that the change set is rather small, spanning only a hundred lines over four files. You then check the history and find that to get to this change set, you applied over 40 commits (this might sound like a textbook example but happens quite often in the real world)! So, rather than merging all these 40 commits and polluting the repository history, you can **squash** these commits into a single commit.

> **Note**: Though squashing sounds simple, which is true if done correctly, it can be a huge pain and disruptive to the team if done without any proper thought. A good example of squashing commits in a branch is when all commits in this branch were added directly to this branch, and NO MERGE was done from another branch to this one. If there were merges and you try to squash the commits, then you cannot blame the authors of this submodule when you wreck havoc in the repository!

Squashing commits in the case illustrated above can be done with the help of the `git rebase` command we looked at earlier.

Here's an illustrative example.

Suppose you have three commits in your current branch with commit hashes `ce45ae`, `df46gr` and `1fcv5o` that you want to squash together into one. We use the `git rebase` command with the `-i` flag to indicate that the rebase is interactive.

```
$ git rebase -i HEAD~3
```

This will open up a text file, called the commit list file, in the default editor (which you specify when installing Git or via the command `git config --global core.editor <editor-name>`).

The file will look similar to this:

```
pick ce45ae Some change
pick df46gr Some other change
pick 1fcv5o One more change

# Rebase 42dcf79b..3d74b79d onto 42dcf79b (3 commands)
```

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

The order of the commits are oldest to most recent, and the last one is the where the HEAD points to at the moment. Except the oldest commit, i.e., ce45ae, change the `pick` statement for the rest of them to `squash`.

What we are doing here essentially, is taking the commits, starting from HEAD, and going back the oldest one (thrid one in this case), while applying the changes in **patches**.

After the changes, the text should look like this:

```
pick ce45ae Some change
squash df46gr Some other change
squash 1fcv5o One more change
...
```

Now, save this file and close it. Once you do, Git will open up another text file which would look similar to this:

```
# This is the commit message #2:

Some other change

# This is the commit message #3:

One more change
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
...
```

We get a chance to modify the new commit that will generated after squashing at this moment. Let it be updated to convey the info about the changes spanning these three commits rather than a message that mentions that the commits were squashed:

```
Introduced blah blah blah which is awesome

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
```

Save and close the file again as before.

```
$ git rebase -i HEAD~3
4 files changed, 831 insertions(+), 625 deletions(-)
 create mode 100644 file1.js
 create mode 100644 file2.js
Successfully rebased and updated refs/heads/my-branch.
```

Now, if we were to run `git log` and check the history on the branch, we would not see the commits we just squashed, instead a new one which has the cumulative changes present in those previous three commits.

## Amending commits

If you have a commit that is present only locally (not pushed to remote yet), you can use the `git amend` command to alter the commit message.

```
$ git commit --amend
```

This will bring up the default the editor specified for Git and you will be prompted to enter the commit message.

If your *last* commit is already pushed to remote and it is the *only* commit whose message you want to alter, then use the above command followed by `git push` with the `--force` flag to update the commit on remote as well. But first checkout the branch which has this commit (if not already on this branch).

```
$ git checkout <your-branch>
$ git commit --amend
...Update commit message...
$ git push --force <your-branch>
```

If you want to ammend multiple commits, then we need to use rebase. As with squashing, use the `git rebase` command with the `-i` flag to bring up

the interactive rebase session. As for the number of commits to rebase, that is dependent upon the age of the recent commits.

```
$ git rebase -i HEAD~5
```

This opens the commit list file on the default text editor specified for Git:

```
pick e499d89 Delete CNAME
pick 0c39034 Better README
pick f7fde4a Changed Foo
pick 56xcv23 Changed Bar
pick 00ef12i Changed Buzz

#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Update the command from `pick` to `reword` for the commits that you want to update the messages for:

```
pick e499d89 Delete CNAME
pick 0c39034 Better README
reword f7fde4a Changed Foo
reword 56xcv23 Changed Bar
pick 00ef12i Changed Buzz
...
```

Save and close the file. This will cause a number of files to open up for altering the commit messages, based on the number of commit messages we are updating. In this case two files. Edit each file, save it and close it.

Then forcefully update remote:

```
$ git push --force
```

### Deleting Commits

The process for deleting, or **dropping** a commit is the exact same as amending a commit, except that when we start the interactive rebase and the commit list file appears, we choose the command `drop` to delete them from the history.

If the commits being dropped are already on remote, make sure to forcefully update the history:

```
$ git push --force
```

# DataGrokr Git Assignment Part I

This is part one of three of the Git assignment for new interns at DataGrokr and is a mandatory module for building foundational skills in Git.

> **NOTE**: After completing this part, host the Git repository publicly on GitHub and submit the link to the GitHub repo to your learning coordinator.

## Create a Git repository

Create a Git repository named `my-exciting-project`.

## Create standard branches as per GitFlow.

By default this newly created repo has only a single branch called `master`. Create a new branch off of `master` and name it `develop`.

## Creating feature branches to add changes

Now we want to add some stuff to our empty codebase. Switch to `develop` branch and then create a new branch called `feature/initial-setup`. Create a file called `README` and add the following line as its content: `This is a sample README`. Similarly, create a new file called `LICENSE` and add the following line as its content: `This is a sample LICENSE`.

Also create a Python script with the following contents and name it as `my-awesome-script.py`:

```
#!/bin/python

print('Hello, World!')
```

Now stage the changes and commit it to the current branch.

## Merging changes from working branches to main branches

At this point if you were to list all active branches in the repository, we have the following:

```
$ git branch
* feature/initial-setup
  master
  develop
```

We want to merge changes from our feature branch `feature/initial-setup` (working branch), to the main development branches (`develop` and `master`) like this:

- Merge `feature/initial-setup` to `develop`
- Merge `develop` to `master`

Perform the merge as described above and once done, `master` and `develop` should have the changes made on `feature/initial-setup`.

## Resolving merge conflicts

Switch to the `develop` branch and create a branch called `feature/enhancement-1` off of it. Modify the print statement in `my-awesome-script.py` to say `Howdy, World!`. Stage and commit your changes.

Switch back to `develop` again.

Create a new branch called `feature/my-enhacement-2` from `develop`. Modify the print statement in `my-awesome-scipt.py` to say `Hajimemashite sekai!` (this also means hello world, but in Japanese).

Now, first merge `feature/enhacement-1` to `develop`. And THEN merge `feature/enhancement-2` to `develop`.

## Amending commits

Checkout `develop` and create a new branch called `feature/next-awesome-thing`. Then add the comment `# This is an awesome Python script` to `my-awesome-script.py`, in the line below the *shebang* (the `#!.bin/python`).

Stage and commit your change with the message `Added a comment`. Now, amend this last commit so that the commit message reads `Added an awesome comment`.

## Dropping commits

While still on branch `feature/next-awesome-thing`, drop the last commit that you added.

### Squashing commits

Go back to `develop` and checkout a new branch `feature/part-1-finale`.

Modify the print statement in `my-awesome-script.py` to say `Hello! World, how is it going?`. Stage and commit your change.

Modify the print statement again in `my-awesome-script.py` to say `Hello! World, hope you're doing well!`. Stage and commit your change.

Modify the print one last time `my-awesome-script.py` to say `Hello! World, we are one!`. Stage and commit your change.

Now, squash the last three commits into a single one.

## DataGrokr Git Assignment Part II

This is part one of three of the Git assignment for new interns at DataGrokr and is a mandatory module for building foundational skills in Git.

> **NOTE**: Deliverables for this assignment needs to be submitted as markdown/doc file to your learning coordinator.

### Writing a Good README

The README file in a Git repository is the core of describing what wonders the repository holds.

A well written README has the following characteristics: * It describes WHAT the project is * It describes HOW to setup a local dev environment * It describes PREREQUISITES if any are relevant

That's it. READMEs are not places to cover HOW the code works (the source code and documentation exists for that!), or who uses it.

The "what" part of the description generally covers the functionality the repository provides. It is brief and concise.

> **Note**: This is a "guide" for a good, well written README. There is no magical template that fits all solutions.

#### Practice

Suppose there is a repository called 'country-capital-api' that houses the code for an API written in Python and Flask that returns the name of the country if a capital city is provided. The ASGI server Uvicorn is used to serve this API. Also, there is a microservice endpoint for this API available at `https://example.com/country-capital/<query-params>` that can be used by any project in the organization to consume this API.

Write a README for this repository.

## Using .gitignore to Keep the Repository Clean

What is and what is not "source code" in 2020 is a debate that can start a flame war. But it is important to note that not *all* files and artifacts are meant to be stored in the remote repository. A good example of this is API keys. Suppose that the repository to which you contribute allows different API keys to run the application based on environment type: local or hosted. Local environment means when you run the application on your development environment and hosted means when it runs on a public cloud provider, e.g., AWS. The API keys that are generated for testing locally are either a single key shared by the entire team, or unique per developer and are generally provided by the organization using an identity provider. In a nutshell, you need this API key for the local environment when developing the application (else how will you test your changes?) but at the same is "risky" to store them in the repository. Why is it risky? It is because based on the access level the API key provides, if it falls in to the wrong hands, it could cause the entire identity provider for the orgainzation to be compromised (remember, if this API key grants unrestricted access, anyone with access to this API key can impersonate requests on behalf of this person). This is especially true for organizations that maintain public projects on hosting providers like GitHub. There are black hats who scrape GitHub and other hosting providers in the hopes that they could find exactly one careless person like this. We don't want that to happen, do we?

This example also holds true for things like database credentials, which if checked into the repository, could cause both financial damage (blackhats deleting records on a production table that was not backed up with the latest data), and also a blow to the enterprise's reputation.

The `.gitignore` file exists precisely for that - to "ignore" certain files from being tracked.

Typical repository artifacts are added to `.gitignore` to prevent them from being tracked:

- Binary builds of the source code
- Environment variables file (e.g., `.env` files)
- Temporary files generated while testing locally
- **Files containing credentials**

GitHub maintains an official list of `.gitignore` files for various languages/frameworks: https://github.com/github/gitignore.

For more details, please check a more detailed overview of `.gitignore` here: https://www.atlassian.com/git/tutorials/saving-changes/gitignore.

### Practice

Consult the `.gitignore` example repository linked above, and write a `.gitignore` file for a repository to ignore the following repository artifacts:

- All files inside the `build` directory at the root of the repository
- All files ending with `*.env`
- All files in the subdirectory `test-runs/logs`
- All CSV files anywhere in the repository

## Raising Clean Pull Requests

As mentioned in the "Branches" submodule, we rarely directly merge changes from our feature or bug fix branches to the main development branch. This is because all changes typically are subject to the review of the entire team to ensure quality and consistency in the code base.

In actual practice, you are required to branch off the main development branch, add changes to your branch, and then raise a PR from your branch to the main development branch. The ability to raise PRs is something that all hosting providers like GitHub provide.

A typical PR consists of the following:

- PR title & description
- Source branch
- Destination branch
- A file/tree view of files that changed
- Line by line differences for each file

Since not all members of the team have the time and energy to go through each line of change, it is important to raise clean PRs.

### Things to keep in mind while raising a PR

- Short and descriptive title The PR title should mention the story ID and a short one line version that encapsulates the story
- WHAT the PR changes This is usually the PR description. Here we briefly describe WHAT has changed. We don't describe HOW it affects things because that is described in the change set already. We summarize the changes made in a few lines, and at most a paragraph.
- WHY are the changes required This follows the WHAT block in the PR description and describes WHY we would like this change. For example, a customer has requested the ability to download reports in addition to view them and if the PR covers this feature, then this should be in the WHY block.
- Any notes that changes workflow Sometimes, certain changes might alter certain aspects of the development. For example, if the local development environment were to be containerized, then the original build scripts will probably no longer work and the method of invoking them would change as well. If changes like these are a part of your PR, then then should be highlighted.

- Raise multiple small PRs It is highly recommended to raise small, but numerous PRs when working on a feature or bug, if possible, rather than one single but large one that spans twenty files and hundreds of lines of changes. It is easier for your teammates to review your changes and give feedback.

This is a good template for raising PRs:

```
PR Title: (feat|bug|hot|chore)/<JIRA story ID>: <Short concise title>
PR Description:

WHAT:
A short paragraph of what changes are introduced in this PR.

WHY:
A short description of why these changes are needed.

Notes:
Any detail that needs to be highlighted goes here.

PR Reviewers:
Add your teammates, including the lead.
```

### Practice

Suppose that the repository you are currently working on contains code for a front-end application that allows customers to get notified via email about updates made to product listings. A certain percentage of customers now want the ability to specify there WhatsApp number as a means of getting notified as well in addition to email notifications. You have been assigned a story with ticket ID FEAPP-420.

FEAPP-420 reads like this:

```
Title: Allow customer notification via WhatsApp
Decription:
Customers have been requesting the ability to get
notified about product updates via their WhatsApp number.

This mode of notification is optional, but if specified, should cause
notifications to go out both on their emails as well as WhatsApp numbers.
```

Suppose that you created a branch off of the main development branch, and are now raising a PR to the main development branch. Write a PR description for this.

## Review and Approval Best Practices

Software engineering in enterprises is a collaborative effort. Rarely is the case that a single developer works on a single repository. It is important to keep in mind that the goal of the team, when reviewing code, is to ensure quality of the code base while adding features at a rapid pace.

These are some of the things to expect as review comments from your teammates when they are reviewing your PRs:

- Coding style: make sure the indentation, casing and naming conventions are consistent with what the team agreed to keep.
- Grammar: It is easy to misspell or use incorrect sentences in comments in the code from time to time. If people are pointing this out, it's not a personal insult but just to ensure consistency.
- Programming best practices: This is a very very broad term, but your team might have certain standards and best practices when implementing things. If your implementation does not meet their standards, they might comment on your "design" and "approach". Again, this is not a personal, but to ensure quality of the codebase.
- Nitpicking: Sometimes small, obscure things in the code can be called out. For example, you might prefer to add spaces in between separators (`const int i = 4;` vs `const int i=4;`), and comments might cover these. Since these are nitpicks, and depending upon the rapport with your team, you can choose to polietely disagree.
- Wait time for getting approvals: Your teammates, just like you, have to complete things on their own plate, in addition to reviewing your code, so keep in mind that the feeling that they are making you wait is not to be personally taken. There is a lot of human factor into this, so patiently wait, and occasionally nudge them politely if the changes are time critical.

Just as the above points are true for people reviewing your code, it is true for you reviewing their code as well. Always try to be specific with your comments.

Some guidelines:

- Don't be vague. For example, `this block looks messy, can you clean it up?`. This offers no insight to the developer who raised the PR and they might simply ignore you.
- Agreeing to Disgaree: Sometimes you might have a wonderful alternative to what has been implemented in a PR. In that case, bring it to the attention of the person, and if the alternative you propose is indeed better, have a chat with them to discuss it through. If their implementation in no way compromises the quality and performance of the code base, and they don't agree to go ahead with your approach, let them be.
- Use of proper tone: Use "please", "could you", "would you", etc. and be as polite as possible. For example, "Can you please reconsider splitting this function into two?" over "Split this function into two."

- Being nitpicky: This is the counterpoint of the point of nitpicking mentioned above: Just like you don't like a nitpicky reviewer, don't be one to your team either!