What is Git

GIT is an open source distributed version control system created by Linus Torvalds, the creator of Linux, with the goals of speed, data integrity and support for distributed, non linear workflows.

> **Naming**
> Named after the British slang git meaning "unpleasant person". Torvalds said: *"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."*. The readme of the source code says the following:
>
> "git" can mean anything, depending on your mood.
>
> - Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
> - Stupid. Contemptible and despicable. Simple. Take your pick from the dictionary of slang.
> - "Global Information Tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
> - "Goddamn idiotic truckload of sh*t": when it breaks.

Essentially it helps you keep track of changes made to your code and facilitates collaboration on the same codebase.

Setup

Linux comes with git already installed. If you are windows you may need to install it. to check whether it is installed run the following command in the terminal:

```
git --version
```

Even if it is installed you may wish to reinstall it if it was installed a while ago in order to set some more useful defaults. For example, GitHub uses main as the default name for the master branch for some progressive reason, whereas git uses master. In the new installer there is an option to change that.

> **Info**
> Alternatively, if you have git installed, don't wish to reinstall it but do want to change the name of your default branch when starting new repositories to main you can use the following command:
>
> ```
> git config --global init.defaultBranch main
> ```

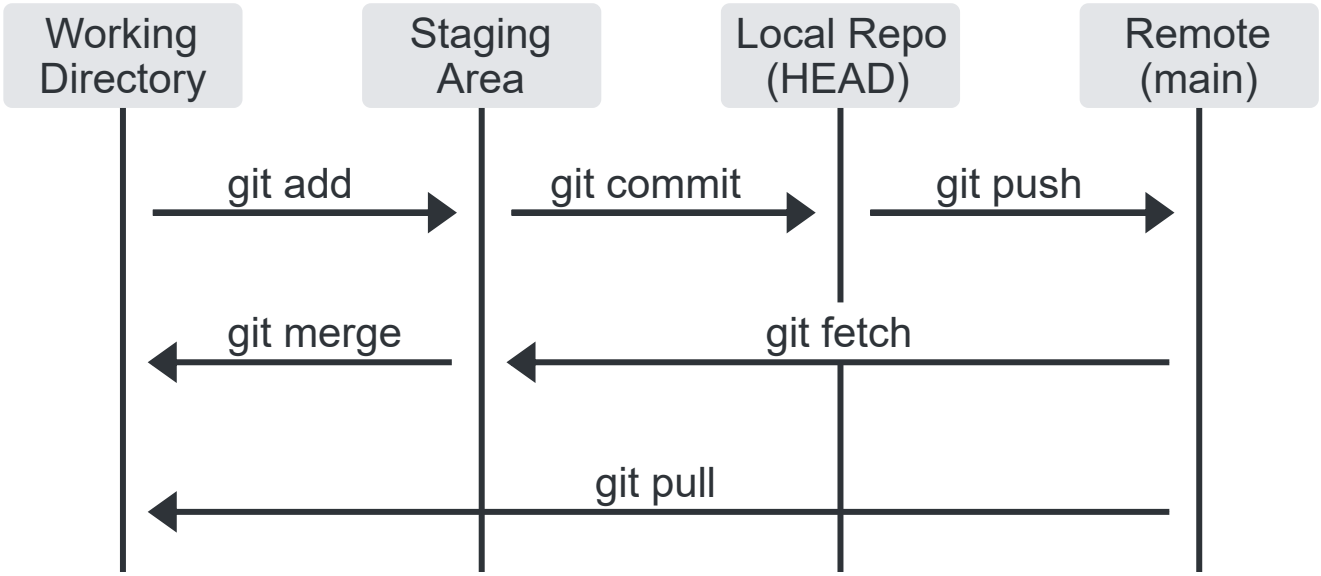
Installation

1. Go to https://git-scm.com/download/win and click the first link to download the installer. Run it once installed.
2. On the **Select components** page make sure the box for Git Bash is checked.
3. On the **Adjusting the name of the initial branch in new repositories page** select "override the default branch name for new repositories" and type `main` in the text box.
4. Continue with the defaults for the rest of the windows and install.
5. Now that git is installed you need to set your credentials. These are used for all commits that you make:

   ```
   git config --global user.name "<your name>"
   git config --global user.email "<your email>"
   ```

   These have been set globally on your device. If you wish to use different credentials for a specific repository run the commands again in that repository without the `--global` option.

Basics



Very basically, git allows you to track the changes you make in your *working directory* by comparing them to the last committed version.
To start using git to track your changes you need to initialize your repository. A repository is just a collection of code files, a folder containing your code. This is done with the command

```
git init
```

> Behind the scenes this creates a folder called `.git` that contains all the files required to keep track of your changes. The . at the beginning of the name is because by default Linux ignores

files and directories that start with .

The convention is to create all of your starting files, your README, .gitignore files etc and then create an initial commit with the message "initial commit". First add the files with the command:

```
git add [list of file names, or . for all]
```

Once a modified file is staged it can either be committed or modified again, requiring it to be re-staged. When you wish to commit your changed you can run the following command to commit all staged files to your local repository:

```
git commit -m "<commit message>"
```

Each commit you make is like a checkpoint that git can go back to at any point, should you wish to do so.

If desired you can commit directly without staging by including the `-a` flag, which will automatically stage all files that are currently being tracked (files that were created before the last commit). However skipping staging generally isn't recommended.

> **.gitignore**
>
> The .gitignore file is a special file used to tell GIT which files to ignore and not track. You want this if for example you have an env file with local configurations that you don't want to push to a remote repository for others to see, or if for example you are using something like NPM which installed packages directly in your working directory (in a folder called node_modules).
> .gitignore is a simple text file that takes blob patterns to ignore, one per line.
> `node_modules` for example would ignore all files and folders called `node_modules`
> `node_modules/` would ignore only folders with that name.
> `*.js` would ignore all files ending with js.

Branches

In order to facilitate simultaneous and asynchronous development git has the concept of branches. A branch starts of as an exact copy of the codebase, evolving as you work on it. When you start a repository it has a single branch, called main.
To create a new branch you can use one of the following commands. The checkout command is used to switch between branches:

```
# create branch from the current branch and check into it
git checkout -b <new branch name>
```

```
# create a branch from another branch and check into it (f)
git branch <new branch name> <base branch>
git checkout <new branch name>
```

> ⚠ **Warning**
>
> When you switch branches any **uncommitted changes will switch with you**. They will not remain on the branch you were working on. This can cause significant issues, so it is recommended to either commit before switching branches or use git stash.

Branching allows you or multiple people to work on different parts of the project at the same time without affecting the main branch. When you complete what you were doing on your branch you may wish to merge it with the main branch once again. You will need to commit your changes on the branch, switch to the branch you wish to merge into using `git checkout` and then use the following command:

```
git merge <new branch name>
```

> ⓘ **Info**
>
> The merge command is not the only command to combine branches. See Rebase.

If there are no conflicting changes, different changes to the same code, the branch will be merged, rendering them identical. You can now delete the branch using

```
git branch -d <new branch name>
```

If there are conflicts you will need to review them, deciding what to keep. Visual Studio Code makes this very easy.

When working with other people you may wish to review their changes before they are permitted to merge their branches into main. This is where pull requests come in.

> ⚠ **Warning**
>
> When working on larger projects it is a good idea to have a development branch instead of making changes directly to main. Imagine you have a product that is being used and a bug is discovered, but you cannot patch it without also releasing the half finished feature you have been working on because you have already made multiple commits to the main branch.
> There are multiple common branching strategies to solve this issue, more information can be found here.

Remotes

GIT facilitates collaboration through the concept of remote repositories. Just like your local repository is a copy of the code base on your device, a remote is a copy of the code base on a remote device that you and others can push to and pull from. You can host your own remote server, but the most commonly used remote service is called GitHub, owned by Microsoft. I will be using that in all further examples.

To start using your own remote repository you first need to create it. It needs to be created independently from the repository on your computer and requires a free GitHub account. Once you have an account create a new repository. Do not check any of the boxes to add .gitignore, license or readme, as if you do it will require you to pull the changes from the remote before you can start commiting.

To add this new remote as a new remote for your local repository you need to use the following command:

```
git remote add origin https://github.com/<github username>/<repo name>.git
```

origin is just an alias for this remote, it can be named whatever you want, but origin is the convention. Should you wish to add multiple remotes you would need to choose different names.

If the remote you have added already have files, if you checked and of the boxes when it was created or if it was already an existing repository you will need to use the following command to merge the files from the remote with those in your local repo:

```
# this is assuming you wish to merge with the main branch
git pull origin main
```

Important

Before you can push to a remote repository you need to authenticate yourself. In order to do this you will need to create a personal access token on GitHub (see GitHub Credentials for why this is better than a password and for alternate auth methods).

1. Log into GitHub
2. Click on your icon in the top right corner
3. Click Settings
4. Click Developer settings on the left
5. Click Personal access tokens on the left
6. Click Generate new token
7. Set the expiration to 1 years time
8. Check the `repo` scope under Select scopes

9. Click Generate token at the bottom
10. Copy your token, as you won't be able to see it again

Before you run the next step, which will ask you for a username and password (use the token you just generated as a password), run the following command. It will store your credentials in a file called .git-credentials in your home directory and use them instead of prompting you in the future.

```
git config --global credential.helper store
```

NOTE: These credentials are stored ***in plain text*** on your computer. If this is not okay with you it is advised you use a third party option, `credential.helper cache` or input your details every time.

When you are ready to push your changes to the remote repository you can do so with the following command

```
git push -u origin main
```

This sets the default remote for the main branch as origin, and pushes the changes to the main branch on the remote. From this point on if you wish to push main you only need to write `git push`.

Fetching changes

There are two ways to update your repository with changes made to a remote repository, git fetch and git pull.

Git Fetch

Git fetch is the less aggressive option. When you fetch a remote repository a new branch is made in your local repo for every branch in the remote. If you fetch a single branch a new branch is created for that branch. The name of the branch is `<remote name>/<branch name>`.

```
git fetch [remote] [branch]
```

If no remote is specified the default remote is used, if you specified one.

You can checkout the new branches just like you can a regular branch, but you will be put into ***detached head mode***. This is essentially read only mode. You can still make changes and even commit them but they will not affect the remote branch. If you do make changes and want them to be saved you will need to create a new branch to save them to with the following command:

```
git switch -c <new branch>
```

If you want to merge the branch you pulled into your corresponding branch switch to your branch and use

```
git merge <remote>/<branch>
```

> **Info**
>
> It is possible to switch to an old commit as if it was a branch using the command
>
> ```
> git checkout remote/branch
> ```
>
> Just like remote branches, checking out an old commit also puts you in detached head mode.

Git Pull

```
git pull [remote]
```

Git pull immediately updates your local repository to match a remote repository (if it is ahead of your local one) by first calling `git fetch`, followed by `git merge`, creating a new merge commit.

If you have uncommitted changes the pull will abort and request that you either commit or stash your changes.

Cloning

Cloning allows you to copy the contents of a remote repository into a new local repository.

```
git clone <remote url>.git [directory]
```

If you do not specify a directory to clone into one will be created with the name of the remote repository. You can specify . as the directory to clone into the current repository.

Cloning is done by downloading the initially committed files and then applying all the changes made in subsequent commits. If you only wish to download the latest version without all the history you can use the `--depth 1` argument.

Pull Requests

Remote repositories allow multiple people to push changes to the same code base, but you may not always want this happening without some idea of what it is they are trying to change. This is where pull requests come in. A pull request is a request to merge a branch into another branch, allowing you

and others to review the proposed changes and make suggestions. GitHub provides a convenient interface for this. Pull requests also allow you to run automatic checks on the code in the form of actions, such as enforcing coding and styling conventions, or checking that it passes all your current tests to make sure the new code doesn't break any existing features.

Forks

What happens if you wish to contribute code to a project that you don't have access to? You can't just create your own branch. This is what forks are for. You can fork any public repository, which is essentially copying the codebase. You can then do what you like with the code, assuming you follow the license. If you then wish to request that the owner of the original project merge your changes you can create a pull request from your fork, requesting that it be merged into the original. Automated tests will run, and members of that project will review your code, request changes if necessary, and if they approve they will sign off on it. GitHub allows you to set how many members on a project are required to review pull requests before they are allowed to be merged. This is how one contributes to open source projects.
As you can see, the misinformed myth about open source projects being dangerous because anyone can change then is indeed a misconception. Anyone can request to make a change, but the only changes made are those that have been scrutinized.

Rolling Back Changes

What happens when you discover that after a recent commit some of the existing features of your product no longer work as intended? Finding and fixing bugs takes time, and its generally not a good idea to leave bugs in the wild any longer than you need to. What you can do is roll back the production branch to a point where you knew it was working, then fix the issue without a rush.
There are numerous ways to do this.

Revert

```
git revert <commit to revert to>
```

The revert command will revert all the changes after the specified commit as a new entry in the version history.
You specify the commit you wish to revert to using the commit ID, which can be seen by running `git log`. You do not actually need to specify the entire ID string, just enough of it to make it unique from the others, 6-8 characters is usually enough.

> **Info**
>
> You can also use the word `HEAD`, which is an alias for the most recent commit.
> `HEAD~<n>` is the the n^{th} commit before that.

Reset

Reset differs from revert in that instead of changing back as a new entry in the version history, reset removes entries from the version history. Just like revert you can also use the commit hash or an offset from `head`.
There are two forms or reset, hard and soft.
Soft reset

```
git reset head~<n>
```

Soft reset is the default. It moves all changes between the current commit and the one you are resetting to to the staging area.
Hard reset

```
git reset --hard head~<n>
```

Hard reset is dangerous as it cannot be undone. This will permanently remove all changes after the specified commit. There are few instances where a hard reset is necessary.

Stashing Changes

Stashing allows you to temporarily store all the changes you have made to a particular branch since the last commit, to be retrieved later. This is useful for a number of reasons. You can stash your current changes, both staged and unstaged with the command

```
git stash
```

You can also annotate a stash. This is useful when you have multiple stashes, as it makes it easier to keep track.

```
git stash save "<message>"
```

To unstash your changes you can use one of the following commands:

```
# apply stashed changes and remove them from the stash
git stash pop
# apply stashed changes but leave them in stash
git stash apply
```

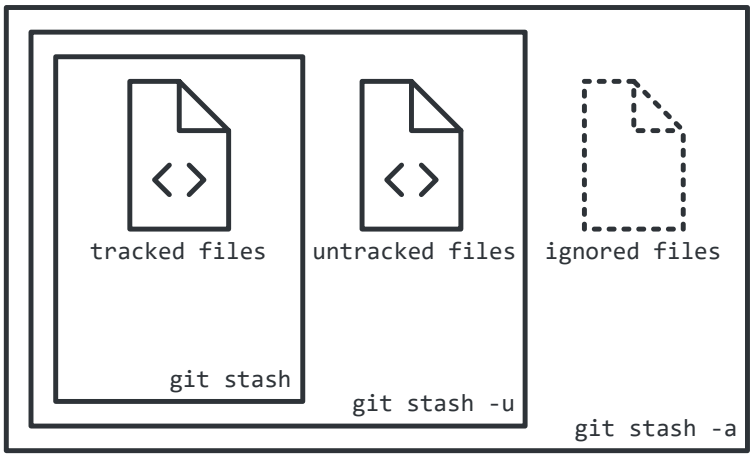
It is important to note that by default stash ignores all files untracked files, files that have been created after the last commit and have not been staged. It also ignores all files ignored by the .gitignore file.

To make stash stash files that are not yet being tracked you can use he `-u` flag:

```
git stash -u
```

To make stash stash all files, including untracked and ignored files use the `-a` (`--all`) flag:

```
git stash -a
```



> **Info**
>
> The most common use for stashing is when you are working on a branch, need to switch to another branch but are not ready to commit yet.
> Another common use is when you wish to move the changes you are currently making from one branch to another. Perhaps you accidentally started working on the main branch and wish to move your progress off to a development branch.

Multiple Stashes

You can stash multiple times and retrieve a specific stash by referencing it's number:

```
git stash pop stash@{n}
```

where n is the number of the stash you wish to pop.

If you want to see all the current stashes you can list them with the command

```
git stash list
```

If you annotated your stashes when you saved them the annotations will show up here.