You are welcome to go through our tutorials but please keep in mind that as we rely on community PRs for maintainance they may be out of date.

Introduction to the Git command line

PREREQUISITES:

- Basic understanding of the system command line (Introduction to the Command Line)
- Git installed and GitHub account set-up (Get set-up with Git and GitHub.)

Introduction to the Git command line

The following set of examples will help you understand how to use Git via the command line, while carrying out common day-to-day tasks.

Example 1: Everyday commands

Create and add your project in Git

To start, let's make a directory to store our files and initialise the directory as a Git repository on our local system. This is the same as we did in the Set-up tutorial previously.

```
$ mkdir practising-git
$ cd practising-git
$ git init
```

> You may recall this will make the directory, change directory so we're located in it, then initialise it as an empty Git repository.

Create a file.

We can create a new file in our local directory and add a line to it at the same time, with the below.

```
$ echo "<h1>Learning git</h1>" > index.html
```

> The above command will output `<h1>Learning Git</h1>` and store it in `index.html`. Open up the file and have a look.

Check the Git repository status

To check the state of our working directory and the Staging area (where we hold files until we commit them), we use the following command.

```
$ git status
```

> The above command will tell you which files in the current directory have been changed and staged, which haven't and which files aren't being tracked by Git.

Add your file to the local repository and commit your changes

When we create new files or change existing ones, we 'add' them to an index of items to be committed to the local repository. The 'commit' command saves our changes into the local repository.

```
$ git add .
$ git status
$ git commit -m 'this is my first command-line commit!'
```

> `.` will add **all** the files in the current directory and subdirectories. You should only use it when initialising your repository. The rest of the time it's recommended that you specify the individual file names to be added.

Check the Git commit history

To check a list of commits in a repository use the `log` command.

```
$ git log
```

> Don't forget, you can hit `q` to quit out of a log if it's too long.

By default, with no arguments, git log shows the most recent commits first.

If we want to limit the number of log entries displayed we can use a number as an option, such as -3 to show only the last three entries.

```
$ git log -3
```

A very useful option is the –patch or -p. This command shows the difference of each commit. This can be very helpful to see what has changed on the last commited changes.

```
$ git log -p
```

If we want to see a few stats for each commit we can use the –stat command.

```
$ git log --stat
```

With the –graph command we will be able to draw a graphical representation of the commits. It will display an ASCII graph of the branch and merge history beside the log output. We can use this command with the –oneline command to display the commits in an alternate format in one line.

```
$ git log --graph --oneline
```

Transferring files from our local project repository to an online service

Before we can add files to a remote repository, we need to have created an account with a service that is hosting that repository. If you've not done that yet, head over to our tutorial: Get set-up with Git and GitHub.

If you've already done that, then great - just run the commands below, adding in the correct remote repository URL, such as `https://github.com/codebar/tutorials.git`.

```
$ git remote add origin <repository-url>
$ git push -u origin master
```

> This is worth repeating: We first add the location of a remote repository, in our case the remote repo is on Github and we've callled it 'origin' as it's the original repository we cloned down to our system. Then we 'push' our changes to the origin's 'master' branch. When there, we can raise a new 'Pull Request' (PR) to get the changes 'merged' into the live code. Check out 'Get set-up with [Git and GitHub]'(../set-up/tutorial.html) tutorial for full details around this.

What is `remote`?

`remote` is the URL of your repository in any online repository hosting service, such as GitHub. The command `git remote` lists all the remote repositories you have configured. You could have the same repository stored in many places like GitHub and GitLab or Heroku and in such cases you will add a remote repository reference to you local machine, as we did above, for each of the remote repositories you have. `remote` is the URL of your repository in any online repository hosting service, such as GitHub. The command `git remote` lists all the remote repositories you have configured. You could have the same repository stored in many places like GitHub and GitLab or Heroku and in such cases you will add a remote repository reference to you local machine, as we did above, for each of the remote repositories you have.

The structure of the command to add a new `remote` is `git remote <add|remove> <name of remote> <url of remote>`.

List all your remote repositories

`git remote`

Or to see more information you can use the verbose (-v) flag

`git remote -v`

Syncing your local copy with the remote copy

```
$ git pull origin master

Username for 'https://github.com': <your username>
Password for 'https://<username>@github.com': <your password>
```

> When you are working with a remote repo it is important to sync your local repo before doing any commit, merge or push.

Syncing the remote copy with your local copy

```
$ git push origin master
$ git log
```

Example 2: Working with a remote service

Update the `index.html` file and then commit and push the changes

```
<html>
    <head>
        <title>Learning Git!</title>
    </head>

    <body>
        <h1> Learning Git </h1>
          <dl>
            <dt>Initialise a Git repository</dt>
            <dd>git init</dd>
            <dt>Add files to Git</dt>
            <dd>git add filename</dd>
          </dl>
    </body>
</html>
```

index.html

Check the status of your repository

```
$ git status
```

Commit and push the changes

```
$ git add index.html
$ git commit -m 'updated to include the commands I learned today'
$ git push origin master
```

Check the repository Git history

```
$ git log
```

Check your code online (from the GitHub or GitLab website)

Example 3: Verifying changes before any commit

Update `index.html`

```
<html>
    <head>
        <title>Learning Git!</title>
    </head>

    <body>
        <h1> Learning Git </h1>
          <dl>
            <dt>Initialise a Git repository</dt>
            <dd>git init</dd>
            <dt>Add files to Git</dt>
            <dd>git add <filename></dd>
            <dt>Checking file changes</dt>
            <dd>git status</dd>
          </dl>
    </body>
</html>
```

Check the changes

```
$ git status
$ git diff
```

The -/+ indications you can see mean

- indicates lines removed from the code.

+ indicates lines added to the code.

```
$ git diff

diff --git a/index.html b/index.html
index 21f15d1..c2031f1 100644
```

```
--- a/index.html
+++ b/index.html
@@ -10,6 +10,8 @@
              <dd>git init</dd>
              <dt>Add files to git</dt>
              <dd>git add <filename></dd>
+             <dt>Checking file changes</dt>
+             <dd>git status</dd>
            </dl>
      </body>
 </html>
```

After you verify your change, commit and push them

```
$ git add .
$ git commit -m 'Added git status description'
$ git push origin master
```

Example 4: Discarding uncommitted changes

Edit the index.html file and then check the changes.

```
$ echo 'oh no!' > index.html
```

> Have a look at changes to the file using `git diff`

Check the status of the repository

```
$ git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   index.html

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

To discard the changes checkout the file

```
$ git checkout index.html
```

Don't forget to verify the changes

```
$ git diff
$ view index.html
```

Example 5: Revert committed changes

Repeat the steps below to change and commit a file

```
$ echo "oh not again" > index.html
$ git diff
$ git add index.html
$ git commit -m 'Oops, I just deleted my list'
```

> Can you explain the commands you just used?

Check the log history

```
$ git log

commit aafbe36777e19244ba5030cbc9467244a7163b61
Author: Jane Doe <jane@codebar.io>
Date:   Tue Jun 3 21:12:57 2014 +0100

    Oops, I just deleted my list

commit dbb313d28de82c11535968584ce2e149b1fc74ad
Author: Jane Doe <jane@codebar.io>
Date:   Tue Jun 3 21:06:09 2014 +0100

    Added git status description

commit c0bb15bf9f75613930c66760b90b2ccc1af0d2d6
...
...
```

Resetting the last commit

```
$ git reset HEAD^

Unstaged changes after reset:
M index.html
```

> The caret (^) after HEAD moves head back through commits. HEAD^ is short for HEAD^1 and in the same way you can apply HEAD^2 to go back two commits ago.

Check the log again

```
$ git log
```

> Did you notice that the last commit is no longer there?

Now check the status and discard the changes in the file

```
$ git status
```

> Do you remember how to discard the changes? Have a look earlier in the tutorial.

Example 6: Revert committed and pushed changes

You can correct something you pushed accidentally by changing history. In the following example you will see how can you revert the last pushed commit.

Run the following steps

```
$ echo "this change will be soon reverted" > index.html
$ git diff
$ git commit -am 'add another broken change'
$ git push origin master
$ git status
$ git log
```

> `git commit -am 'commit message` is short form for `git add .` followed by `git commit -m 'message'`.

> What does `git push` do?

Reverting a commit

```
$ git log

commit f4d8d2c2ca851c7317664110917278048 7da9c1d
Author: Jane Doe <jane@codebar.io>
Date:   Tue Jun 3 21:17:57 2014 +0100

    add another broken change

commit dbb313d28de82c11535968584ce2e149b1fc74ad
Author: Jane Doe <jane@codebar.io>
Date:   Tue Jun 3 21:06:09 2014 +0100

    Added git status description

commit c0bb15bf9f75613930c66760b90b2ccc1af0d2d6
...
...
...
```

You need to grab the commit identifier and then revert to it

```
$ git revert f4d8d2c2ca851c7317664110917278048 7da9c1d
```

After reverting the changes, you have to push the code to the remote repo to apply them

```
$ git push origin master
```

Extras

Creating a Git Config file

Git allows us to define configuration settings that affect either just the repository we're working with, such as the URL of the remote repository location, or global settings such as Aliases for common commands (see below). There's a great example over at https://gist.github.com/pksunkara/988716

Create a file called `.gitconfig` in the root directory (parent folder) of your local Git repo by typing `git touch .gitconfig` in the terminal window. Though it may look odd, this file doesn't have an extension such as `.txt` like typical files. Now let's practise modifying this file by adding the following configuration items.

User name and email

If you didn't add your name and email address in the Set-up tutorial, add them now. These are added to each Git commit so it's clear who made the commit to the repo.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "name@domain"
```

If you want to just edit the `.gitconfig` file directly then look for it in your root folder, open in your favourite text editing software add the following:

```
[user]
    name = "Your Name"
    email = "Your email address"
```

Creating shortcuts (aliases)

In order to save time and key strokes when working on the command line in a terminal window, you can create aliases for your most commonly used Git commands. Here are some examples.

```
$ git config --global alias.cm 'commit -m'
$ git config --global alias.co checkout
$ git config --global alias.st status
```

To use them just type `git cm "Message here"` or `git st` for example. Simply put, use the alias in place of the full text Git command, then add any switches on the end (such as a commit message) as you would normally. If you want to just edit the `.gitconfig` file directly

then you would add the following section:

```
[alias]
    cm = commit --message
    co = checkout
    st = status
```

> Can you think of another command that you would find handy to shorten down?

Telling Git to try and fix whitespace issues before committing

A common issue when editing files, especially if the file has been worked on using both Mac and Windows systems, is having a) a trailing whitespace at the end of a line of code, b) a line that is entirely whitespace or c) having a leading whitespace before a tab. When performing a commit where a file has these, Git will prompt you about how (and if) you'd like to fix them. We can tell Git to always fix whitespace issues by adding the following to our `.gitconfig` file:

```
$ git config --global apply.whitespace fix
```

If you prefer to just edit the `.gitconfig` file directly, then you would add the following section:

```
[apply]
  whitespace = fix
```

Ignoring files across directories (.gitignore)

Very often you have certain types of files or directories in the folder where your code and other assets are. When we commit our code to the Git repository every file will be commited and added to the repository. If you want to exclude certain files on your system from being included in the Git repository, we can add a `.gitignore` file to do this. Make sure this file is included in your repo, to share the ignore rules with others.

Firstly, look for the `.gitignore` file in the root of the directory you're working in. If you have one, skip to the next step, if not let's make one now.

```
$ git touch .gitignore
```

To make a reference between the `.gitconfig` file we were working with and the `.gitignore` file we just made, execute the following command:

```
$ git config --global core.excludesfile ~/.gitignore
```

If you prefer, to just edit the `.gitignore` file directly, then you would add the following section:

```
[core]
  excludesfile = ~/.gitignore
```

Within the `.gitignore` file you can add specific files or extensions that you always want excluded. Here's some common examples that you may want to exclude.

```
    *.DS_Store
    *~
    *.log
    *.zip
    *.pkg
    *.rar
```

> Do you know what these files are? You normally wouldn't want to commit logs or packages. For a more complete list of files you may want to exclude have a look at https://gist.github.com/octocat/9257657.

Prettify your log history

It's possible to make your Git logs easier to read by changing the colour and style of them.

On the command line, run the following:

```
$ git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow
```

As previously, if you prefer to edit the `.gitconfig` file directly, add the following as an alias for viewing Git logs

```
  lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr
```

Try it out by running `git lg` in a terminal window. If the log is very long, you can quit out of the log by hitting `q` on the keyboard.

Bonus

Store commands in your history

On the command line you can use the up and down arrows to cycle back over the Git commands you've typed before. This can save time when retrying certain commands or allow you to fix them without retyping the whole command. We can Add HISTSIZE and HISTFILESIZE to your `.bashrc` file to make sure plenty of commands are stored beyond the default 500.

- **HISTSIZE** is the number of commands stored in memory when you are using the terminal.
- **HISTFILESIZE** is the number of commands stored in memory for future sessions.

```
HISTSIZE=10000
HISTFILESIZE=20000
```

After typing a couple of commands in the terminal to generate some history, try executing `Ctrl` + `R` followed by the command you want to run e.g. `git lg` or just use the arrows on the keyboard to cycle back through them.

> You can see the entire history by running `history`

Further Learning

If you want to continue your learning on Git, then watch this video with Anna Skoulikari on our YouTube.

This ends **Introduction to the Git command line** tutorial. Is there something you don't understand? Try and go through the provided resources with your coach. If you have any feedback, or can think of ways to improve this tutorial send us an email and let us know.

Back to tutorials codebar mainpage