Review: How to commit changes with Git

Let’s review the basics of git

What is Git?

Git is a tool that allows groups of people to **collaborate** on the same code without getting in each other's way.

It's even great to use when you are on your own because it helps you keeps track of your **code's history**.

What is Git?

It's what we call a **distributed version control system**.

It's a *version control system* because it keeps your code's history as a series of checkpoints or versions.

What is Git?

It's what we call a **distributed version control system**.

It's *distributed* because everybody has a fully fledged copy of the entire history of the code. It sounds like that's overkill but it turns out it's really convenient and, for most projects, there's really no drawback.

Getting started with Git

If you haven't already, you need to setup your name and email with Git.

Otherwise your commits will not be attributed properly to you.

In your terminal, run these two commands but with your information:

```
$ git config --global user.name "Han Solo"

$ git config --global user.email han@smugglers.sexy
```

Getting started with Git

You will often see dollar signs $ in examples like this. They signify commands that you have to run in your terminal.

You **don't need to include the $** as part of the command.

```
$ git config --global user.name "Han Solo"

$ git config --global user.email han@smugglers.sexy
```

Getting started with Git

Now that Git knows who you are we can set up a **repository**.

A *repository* is a folder that is set up so that Git can keep track of its history.

You can make changes in a repository and then create a checkpoint to save the current changes in the history. In Git, these checkpoints are called **commits**, but we'll review those later.

Getting started with Git

To turn a normal folder into a Git respository, navigate into the folder and run `git init`.

```
$ mkdir new_repo

$ cd new_repo

$ git init
```

You should see a message like this one:

```
Initialized empty Git repository in
new_repo/.git/
```

Getting started with Git

```
$ mkdir new_repo

$ cd new_repo

$ git init
Initialized empty Git repository in
new_repo/.git/
```

That `.git` folder is a hidden folder where the Git repository information is saved.

Commits

Commits are the building blocks of a repository.

A commit is a “photo” of the code

A commit is uniquely identified by a *sha*

- `ef517e2df870ea53ec7a51cc0de01c801d5ab8f0`

Commits

```
ef517e2 - (HEAD) added info about my job.
cb3487a - created CV.
b8bd3b5 - Extend profile info in about page.
9f01a46 - Added info about me
be3eae3 - Initial commit
```

Getting started with Git

The last thing you should know before you start is to use `git status` to see what's going on in your repository.

```
$ git status
```

You should see a message like this one:

```
On branch master

Initial commit

nothing to commit (create/copy files and
use "git add" to track)
```

Getting started with Git

```
$ git status
```

Get used to running this often so that you know what's going on with your repo.

It (and other Git commands) will even tell you if you aren't currently in a folder with a repository.

```
$ git status
fatal: Not a git repository (or any of the
parent directories): .git
```

Making changes

Now that your repo is set up, you can start to make changes to it.

This is the easy part! All you have to do is create or modify files inside the repository folder.

Making changes

Create a file named `important_program.rb` and let's add some important code to it:

```
puts "Hi there! What this program does is really important."
```

Adding changes

Now that we've made a change, let's run `git status` to see what's changed.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

  important_program.rb

nothing added to commit but untracked files present (use
"git add" to track)
```

Adding changes

It says we have a new *untracked file*, a file that isn't yet in the history.

Note that `git status` is making suggestions about what we should do next:

```
Untracked files:
  (use "git add <file>..." to include in what will be
committed)

  important_program.rb

nothing added to commit but untracked files present (use
"git add" to track)
```

Adding changes

To be able to save these changes in the history, we first need to `git add` the file we just created.

This is the first step of making a **commit**, which is like a snapshot in time of our code.

```
$ git add important_program.rb
```

Adding changes

```
$ git add important_program.rb
```

The `git add` command adds the file to a temporary in-between place called the **staging area** or the **index**.

You can add and remove changes you are thinking of *committing* to and from the *staging area*. It's okay because these changes aren't yet a permanent part of the history.

Adding changes

```
$ git add important_program.rb
```

Now, let's do another `git status` to see where that leaves us:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   important_program.rb
```

Adding changes

```
$ git add important_program.rb

$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   important_program.rb
```

Notice how it says *Changes to be committed*. We are ready to make our first commit.

Trick: Stage all changes

```
$ git add .
```

This will stage all changes for commit (even new files, and deleted files)

Committing

The next step is to run `git commit` on our repository that now has pending changes. Changes aren't "baked in" until you commit.

```
$ git commit --message "Add important program"
```

Committing

```
$ git commit --message "Add important program"
```

You must always provide a message with every commit you make. You should use it to give context about what has changed and why.

The easiest way to add a message to your commit is with the `--message` option.

Committing

```
$ git commit --message "Add important program"
```

If you are lazy you can abbreviate `--message` to `-m`.

Anyway, you should see something like this:

```
[master (root-commit) fc31ed1] Add important program
 1 file changed, 1 insertion(+)
 create mode 100644 important_program.rb
```

Committing

Now when we do `git status` we can see that we've got a clean repo. All our changes have been commited to the history.

```
$ git status
On branch master
nothing to commit, working directory clean
```

Looking at the history

Now we've got out first commit. How can we see our repo's history? We use `git log` to see all the entries in the history so far.

```
$ git log
commit fc31ed1b932dc7b3ca31f6b7c82dfc8d9aaa84b4
Author: Han Solo <han@smugglers.sexy>
Date:   Mon Aug 17 17:00:22 2015 -0400

    Add important program
```

Looking at the history

Use the `--summary` option if you want more information about each commit:

```
$ git log --summary
```

With the `--patch` option you can even see the specific changes in each file:

```
$ git log --patch
$ git log --patch <filename>
```

Ignoring files

Create `.gitignore` file to avoid adding certain files to the repository.

Each lines define a path to ignore.

Do it from the start!

```
log/error.log
bin/**/*
config/secret_keys
db/db_backup
```

Recap: Making changes

Let's make another change, just to recap. Out important code needs an important README file. Let's create a README.md file with these contents.

```
Important Program
=================

A very important Ruby program of unfathomable
importance.
```

Recap: Staging changes

Just like before, let's add it to the staging area to commit it to the history and see the status.

```
$ git add README.md

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  new file:   README.md
```

Recap: Staging changes

But wait! We're not ready to commit this yet. I want to add another really important piece of code to the already important program.

After that's commited I want to add the README.

Removing from the staging area

Let's follow the advice of `git status` to remove these changes from the staging area:

```
$ git reset HEAD README.md
```

The `git reset` command can manipulate the staging area. We are telling it to reset the staging area with regards to the `README.md` file.

Removing from the staging area

```
$ git reset HEAD README.md
```

In general, you should pay attention to what `git status` tells you. It always gives you good advice about the options you have to manipulate files in your repository.

Recap: Making changes

Let's open `important_program.rb` and add another really important piece of code. It should now look like this:

```
puts "Hi there! What this program does is really
important."

puts "MODIFY THIS PROGRAM AT YOUR OWN PERIL."
```

Recap: Making changes

Let's do a `git status` now. Looks different, right?

```
$ 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:   important_program.rb
```

Recap: Staging changes

Let's stage these modifications to the program and do another `git status`.

```
$ git add important_program.rb

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   important_program.rb
```

Recap: Commiting changes

Now we can commit our changes to the history and see the results in the log.

```
$ git commit -m "Add important warning to the program"
[master f813c3b] Add important warning to the program
 1 file changed, 3 insertions(+), 1 deletion(-)

$ git log
```

Recap: README

Now we can add and commit the README.

```
$ git add README.md

$ git commit -m "Add important README"
[master 1936b25] Add important README
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
```

Conclusion

You should make it a point to commit early and often.

Try to group related changes into small commits.

Large commits are hard to figure out later.

Remember kids: **ABC**

ALWAYS BE COMMITTING