**Part 3**

**Q. What is GitHub?**

**A.**

* GitHub is a web-based Git repository hosting service.
* GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.
* It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.
* As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.
* In general, GitHub is a combination of, “A Publishing tool, Version Control System, Collaboration Platform”.

**Q. When was it created?**

**A.**

* Development of the GitHub platform began on October 1, 2007.
* It had been made available for a few months prior as a beta release.
* It was finally launched on April 10, 2008.

**Q. Why?**

**A.** GitHub has many advantages, such as,

1. Versioning Project and having Version Control
   * Version Control System (VCS) is used for giving versions to the different release of the projects. VCS also maintains the past history of the code. The codes will be saved and we can look back to check the history of the code. So it’s good for keeping the code clean and well maintained.
2. Publishing projects – After publishing the project, the user can download it, install it, and work with it, and even customize it if anyone wants to.
3. Collaborating projects – If some group of developers are working on a project, even if they are not local, say they are all over the world; they can use it as a sort of a hub and contribute their work, from any place.

**Q. By who?**

**A.** The founders of GitHub are Tom Preston-Werner, Chris Wanstrath, PJ Hyett.

**Q. What similar platforms exist?**

**A.** Similar Platforms are,

* GitLab
* [Bitbucket](http://alternativeto.net/software/bitbucket/)
* SourceForge
* LaunchPad
* Trac
* GitBucket
* CodePlex
* Assembla
* RhodeCode
* CodeBase

**Part 4**

1.1 Git allows groups of people to work on the same documents (often code) at the same time, and without stepping on each other's toes. It's a distributed version control system.

Our terminal prompt below is currently in a directory we decided to name "octobox". To initialize a Git repository here, type the following command:

git init

> git init

Initialized empty Git repository in /.git/

Success!

1.2 Checking the Status

Good job! As Git just told us, our "octobox" directory now has an empty repository in /.git/. The repository is a hidden directory where Git operates.

To save your progress as you go through this tutorial -- and earn a badge when you successfully complete it -- head over to create a free Code School account. We'll wait for you here.

Next up, let's type the git status command to see what the current state of our project is:

$ git status

# On branch master  
#  
# Initial commit  
#  
nothing to commit (create/copy files and use "git add" to track)

Success!

1.3 Adding & Committing

I created a file called octocat.txt in the octobox repository for you (as you can see in the browser below).

You should run the git status command again to see how the repository status has changed:

git status

$ git status

# On branch master  
#  
# Initial commit  
#  
# Untracked files:  
# (use "git add <file>..." to include in what will be committed)  
#  
# octocat.txt  
nothing added to commit but untracked files present (use "git add" to track)

Success!

1.4 Adding Changes

Good, it looks like our Git repository is working properly. Notice how Git says octocat.txt is "untracked"? That means Git sees that octocat.txt is a new file.

To tell Git to start tracking changes made to octocat.txt, we first need to add it to the staging area by using git add.

git add octocat.txt

$ git add octocat.txt

Nice job, you've added octocat.txt to the Staging Area

1.5 Checking for Changes

Good job! Git is now tracking our octocat.txt file. Let's run git status again to see where we stand:

git status

$ git status

# On branch master  
#  
# Initial commit  
#  
# Changes to be committed:  
# (use "git rm --cached <file>..." to unstage)  
#  
# new file: octocat.txt  
#

Success!

1.6 Committing

Notice how Git says changes to be committed? The files listed here are in the Staging Area, and they are not in our repository yet. We could add or remove files from the stage before we store them in the repository.

To store our staged changes we run the commit command with a message describing what we've changed. Let's do that now by typing:

git commit -m "Add cute octocat story"

$ git commit -m "Add cute octocat story"

[master (root-commit) 20b5ccd] Add cute octocat story  
1 file changed, 1 insertion(+)  
create mode 100644 octocat.txt

Success!

1.7 Adding All Changes

Great! You also can use wildcards if you want to add many files of the same type. Notice that I've added a bunch of .txt files into your directory below.

I put some in a directory named "octofamily" and some others ended up in the root of our "octobox" directory. Luckily, we can add all the new files using a wildcard with git add. Don't forget the quotes!

git add '\*.txt'

$ git add \*.txt

Did not add "octofamily/baby\_octocat.txt"

$ git add '\*.txt'  
Success!

1.8 Committing All Changes

Okay, you've added all the text files to the staging area. Feel free to run git status to see what you're about to commit.

If it looks good, go ahead and run:

git commit -m 'Add all the octocat txt files'

$ git commit -m 'Add all the octocat txt files'  
[master 3852b4d] Add all the octocat txt files  
4 files changed, 4 insertions(+)  
create mode 100644 blue\_octocat.txt  
create mode 100644 octofamily/baby\_octocat.txt  
create mode 100644 octofamily/momma\_octocat.txt  
create mode 100644 red\_octocat.txt

Success!

1.9 History

So we've made a few commits. Now let's browse them to see what we changed.

Fortunately for us, there's git log. Think of Git's log as a journal that remembers all the changes we've committed so far, in the order we committed them. Try running it now:

$ git log

commit 3852b4db1634463d0bb4d267edb7b3f9cd02ace1  
Author: Try Git <try\_git@github.com>  
Date: Sat Oct 10 08:30:00 2020 -0500  
  
Add all the octocat txt files  
  
commit b652edfd888cd3d5e7fcb857d0dabc5a0fcb5e28  
Author: Try Git <try\_git@github.com>  
Date: Sat Oct 10 08:30:00 2020 -0500  
  
Added cute octocat story

Success!

1.10 Remote Repositories

Great job! We've gone ahead and created a new empty GitHub repository for you to use with Try Git at https://github.com/try-git/try\_git.git. To push our local repo to the GitHub server we'll need to add a remote repository.

This command takes a remote name and a repository URL, which in your case is https://github.com/try-git/try\_git.git.

Go ahead and run git remote add with the options below:

git remote add origin https://github.com/try-git/try\_git.git

$ git remote add origin https://github.com/try-git/try\_git.git  
Success!

1.11 Pushing Remotely

The push command tells Git where to put our commits when we're ready, and boy we're ready. So let's push our local changes to our origin repo (on GitHub).

The name of our remote is origin and the default local branch name is master. The -u tells Git to remember the parameters, so that next time we can simply run git push and Git will know what to do. Go ahead and push it!

git push -u origin master

$ git push -u origin master

Branch master set up to track remote branch master from origin.

Success!

1.12 Pulling Remotely

Let's pretend some time has passed. We've invited other people to our GitHub project who have pulled your changes, made their own commits, and pushed them.

We can check for changes on our GitHub repository and pull down any new changes by running:

git pull origin master

> git diff HEAD

diff --git a/octocat.txt b/octocat.txt  
index 7d8d808..e725ef6 100644  
--- a/octocat.txt  
+++ b/octocat.txt  
@@ -1 +1 @@  
-A Tale of Two Octocats  
+[mA Tale of Two Octocats and an Octodog  
Success!

1.14 Staged Differences

Another great use for diff is looking at changes within files that have already been staged. Remember, staged files are files we have told git that are ready to be committed.

Let's use git add to stage octofamily/octodog.txt, which I just added to the family for you.

git add octofamily/octodog.txt

$ git add octofamily/octodog.txt

Success!

1.15 Staged Differences (cont'd)

Good, now go ahead and run git diff with the --staged option to see the changes you just staged. You should see that octodog.txt was created.

git diff –staged

$ git diff --staged

diff --git a/octofamily/octodog.txt b/octofamily/octodog.txt  
new file mode 100644  
index 0000000..cfbc74a  
--- /dev/null  
+++ b/octofamily/octodog.txt  
@@ -0,0 +1 @@  
+[mwoof  
Success!

1.16 Resetting the Stage

So now that octodog is part of the family, octocat is all depressed. Since we love octocat more than octodog, we'll turn his frown around by removing octodog.txt.

You can unstage files by using the git reset command. Go ahead and remove octofamily/octodog.txt.

git reset octofamily/octodog.txt

$ git reset octofamily/octodog.txt

Success!

1.17 Undo

git reset did a great job of unstaging octodog.txt, but you'll notice that he's still there. He's just not staged anymore. It would be great if we could go back to how things were before octodog came around and ruined the party.

Files can be changed back to how they were at the last commit by using the command: git checkout -- <target>. Go ahead and get rid of all the changes since the last commit for octocat.txt

git checkout -- octocat.txt

$ git checkout -- octocat.txt

Success!

1.18 Branching Out

When developers are working on a feature or bug they'll often create a copy (aka. branch) of their code they can make separate commits to. Then when they're done they can merge this branch back into their main master branch.

We want to remove all these pesky octocats, so let's create a branch called clean\_up, where we'll do all the work:

git branch clean\_up

$ git branch clean\_up

Success!

1.19 Switching Branches

Great! Now if you type git branch you'll see two local branches: a main branch named master and your new branch named clean\_up.

You can switch branches using the git checkout <branch> command. Try it now to switch to the clean\_up branch:

git checkout clean\_up

$ git checkout clean\_up

Switched to branch 'clean\_up'

Success!

1.20 Removing All The Things

Ok, so you're in the clean\_up branch. You can finally remove all those pesky octocats by using the git rm command which will not only remove the actual files from disk, but will also stage the removal of the files for us.

You're going to want to use a wildcard again to get all the octocats in one sweep, go ahead and run:

git rm '\*.txt'

$ git rm '\*.txt'

rm 'blue\_octocat.txt'  
rm 'octocat.txt'  
rm 'octofamily/baby\_octocat.txt'  
rm 'octofamily/momma\_octocat.txt'  
rm 'red\_octocat.txt'

Success!

1.21 Commiting Branch Changes

Now that you've removed all the cats you'll need to commit your changes.

Feel free to run git status to check the changes you're about to commit.

git commit -m "Remove all the cats"

$ git commit -m "Remove all the cats"

[clean\_up 63540fe] Remove all the cats  
5 files changed, 5 deletions(-)  
delete mode 100644 blue\_octocat.txt  
delete mode 100644 octocat.txt  
delete mode 100644 octofamily/baby\_octocat.txt  
delete mode 100644 octofamily/momma\_octocat.txt  
delete mode 100644 red\_octocat.txt

Success!

1.22 Switching Back to master

Great, you're almost finished with the cat... er the bug fix, you just need to switch back to the master branch so you can copy (or merge) your changes from the clean\_up branch back into the master branch.

Go ahead and checkout the master branch:

git checkout master

$ git checkout master

Switched to branch 'master'

Success!

1.23 Preparing to Merge

Alrighty, the moment has come when you have to merge your changes from the clean\_up branch into the master branch. Take a deep breath, it's not that scary.

We're already on the master branch, so we just need to tell Git to merge the clean\_up branch into it:

git merge clean\_up

$ git merge clean\_up

Updating 3852b4d..ec6888b  
Fast-forward  
blue\_octocat.txt | 1 -  
octocat.txt | 1 -  
octofamily/baby\_octocat.txt | 1 -  
octofamily/momma\_octocat.txt | 1 -  
red\_octocat.txt | 1 -  
5 files changed, 5 deletions(-)  
delete mode 100644 blue\_octocat.txt  
delete mode 100644 octocat.txt  
delete mode 100644 octofamily/baby\_octocat.txt  
delete mode 100644 octofamily/momma\_octocat.txt  
delete mode 100644 red\_octocat.txt

Success!

1.24 Keeping Things Clean

Congratulations! You just accomplished your first successful bugfix and merge. All that's left to do is clean up after yourself. Since you're done with the clean\_up branch you don't need it anymore.

You can use git branch -d <branch name> to delete a branch. Go ahead and delete the clean\_up branch now:

git branch -d clean\_up

$ git branch -d clean\_up

Deleted branch clean\_up (was ec6888b).

Success!

1.25 The Final Push

Here we are, at the last step. I'm proud that you've made it this far, and it's been great learning Git with you. All that's left for you to do now is to push everything you've been working on to your remote repository, and you're done!

git push

$ git push

To https://github.com/try-git/try\_git.git  
3e70b0f..d255a7d master -> master

Success!

1.25 The Final Push

Great! You now have a little taste of the greatness of Git. You can take a look at the wrap up page for a little more information on Git and GitHub, oh, and of course your badge!

Wrap it all Up

**Part 5**

Define the following terms in the context of Git (2 lines maximum):

* **Repository**
  + A directory where Git has been initialized to start version controlling your files.
  + A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and anything the project needs.
* **Commit**
  + On GitHub, saved changes are called commits. Each commit has an associated commit message, which is a description explaining why a particular change was made.
  + Commit messages capture the history of your changes, so other contributors can understand what you’ve done and why.
* **Push**
  + Pushing refers to sending our committed changes to a remote repository such as GitHub.com. For instance, if we change something locally, we'd want to then push those changes so that others may access them.
* **Branch**
  + Branching is the way to work on different versions of a repository at one time.
  + When we create a branch off the master branch, we’re making a copy, or snapshot, of master as it was at that point in time. If someone else made changes to the master branch while we are working on our branch, we could pull in those updates.
* **Fork**
  + A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original.
* **Merge**
  + Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another. This often happens as a Pull Request (which can be thought of as a request to merge), or via the command line.
* **Clone**
  + A clone is a copy of a repository that lives on our computer instead of on a website's server somewhere, or the act of making that copy.
  + With our clone we can edit the files in our preferred editor and use Git to keep track of our changes without having us to be online.
* **Pull**
  + Pull refers to when we are fetching in changes and merging them. For instance, if someone has edited the remote file we're both working on, we'll want to pull in those changes to our local copy so that it's up to date.
* **Pull request** 
  + Pull requests let us tell others about changes we've pushed to a repository on GitHub.
  + Once a pull request is opened, we can discuss and review the potential changes with collaborators and add follow-up commits before the changes are merged into the repository.