Getting Started with Git in a Team Environment

Shaumik Daityari(http://www.sitepoint.com/author/sdaityari/)

May 27, 2014

In a previous article, I discussed getting started with Git (http://www.sitepoint.com/git-for-beginners/), mainly focusing on using Git when working alone. The core philosophy of Git, however, revolves around the concept of a distributed version control system.

The concept of "distributed" means there exist many independent versions of a single project, each with their own history. Thus, **Git is a tool that helps you work with a team that may be geographically distributed**.

We discussed how you could manage your files with Git in the previous tutorial, but the aim was to get familiarized with the concepts of git (add, commit, push). Now that you know how Git works locally, you are in a good position to start using the more advanced features.

Cloning

The first step in the process is to get the code (assuming Git is installed) from a remote resource. `remote` refers to a remote version of a repository or project. We begin by cloning the resource, using the command `git clone`.

```
git clone https://github.com/sdaityari/my_git_project.git
```

A clone URL can be obtained from a GitHub repository's main page, in the sidebar. On successfully cloning the repository, a directory is created (by default, it's the same name as the project). Another interesting fact is that the `origin` remote now points to the resource from which I cloned the repository (in this case, `https://github.com/sdaityari/my_git_project.git`).

Even though Git follows the distributed model, a central repository is usually maintained that contains stable, updated code.

Which protocol Should You Use When Cloning?

Notice that the repository is cloned over the `https` protocol in this example. The other popular choices (http://git-scm.com/book/en/Git-on-the-Server-The-Protocols) that you have are `ssh` and `git`.

Cloning over the `git` protocol requires you to use the origin that looks something like `git://github.com/[username]/[repository]`. This does not provide any security except those of Git itself. It's usually fast, but the big disadvantage is the inability to `push` changes as it gives only read-only access.

If you use the `https` protocol, your connection is encrypted. GitHub allows anonymous pulls over `https` for public repositories, but for pushing any code, your username and password would be verified. GitHub recommends using https over ssh (http://stackoverflow.com/questions/11041729/why-does-github-recommend-https-over-ssh).

The `ssh` protocol uses the public key authentication (http://en.wikipedia.org/wiki/Key_authentication#Authentication_using_Public_Key_Cryptography). This requires you to establish a connection with the remote server over `ssh`. To set up authentication using `ssh`, you need to generate your public/private key pair. In Linux, you run the following command in the terminal to generate your key pair. But if you use the GitHub desktop client, this process is done automatically by the software, and you get an email that an SSH key was added to your GitHub account.

```
ssh-keygen -t rsa -C "[email_address]"
```

Note: in these code examples, I will often include sections within square brackets. Those bracketed sections would be replaced with what's indicated inside the brackets, but with the brackets removed.

You can optionally provide a passphrase that would be needed every time you try to connect. We leave it blank in our case. You can also specify the file where you can save the key. We leave it to the default `id_rsa`.

The public and private keys are stored in the `id_rsa.pub` and `id_rsa` files, respectively. In the case of GitHub, you need to paste the contents of your public key under 'SSH' in your profile.

To check that the process was completed successfully, you can establish a connection to the git user on github.com.

```
ssh git@github.com
```

```
donny@ubuntu:~/my_git_project$ ssh git@github.com
PTY allocation request failed on channel 0
Hi sdaityari! You've successfully authenticated, but GitHub does not provide shell access.
Connection to github.com closed.
donny@ubuntu:~/my_git_project$
```

As you can see, the connection to GitHub was successful but it was terminated because GitHub doesn't allow shell access. This means that you can use the `ssh` protocol to pull or push changes.

An interesting fact is that a `remote` contains the protocol too. This means that connection to a resource through the `https` and `ssh` protocols need to be stored as separate remotes.

Branches

One of the best features of Git is branching. Think of branches as pathways that you take as you progress through your code. These branches can be visualized by the branches of a tree. The only place where our tree analogy fails is that Git also allows you to merge branches.

If you have used branches in other version control systems before, you need to clear your mind as the concepts are a bit different. For instance, if you compare Subversion and Git, their merging mechanisms are different (http://stackoverflow.com/questions/19333088/how-is-svn-v-1-8-branching-merging-compared-to-git).

Imagine you are making a chat application. At one point you decide that you want the snapshots of people to appear next to their name, but that idea hasn't been approved by your boss yet. You create a branch and work on your idea while the old functionality remains intact in a separate branch. If you want to demonstrate your old work, you could switch over to your old branch. If your idea is approved, you can then merge the new branch with the old one to make the changes.

To check the list of branches and the current branch that you are working on, run the following command. It shows the list of branches in your repository, with an asterisk against the current working branch.

```
git branch
```

Creating branches

Git provides a `master` branch, that you work on by default. When creating branches, you must name them properly, just like functions or commit messages. To create a branch, run the following command.

```
git branch [branch_name]
```

This creates a new branch based on the last commit, but remains on the same branch. To switch to a different branch, run the following command

```
git checkout [branch_name]
```

To create a new branch and switch to it immediately, use:

```
git checkout -b [branch_name]
```

This creates a new branch based on the last commit of your current branch. If you want to create a branch based on an old commit, you should append the hash that identifies a commit with the command.

```
git checkout -b [branch_name] [commit_hash]
```

Branching Models

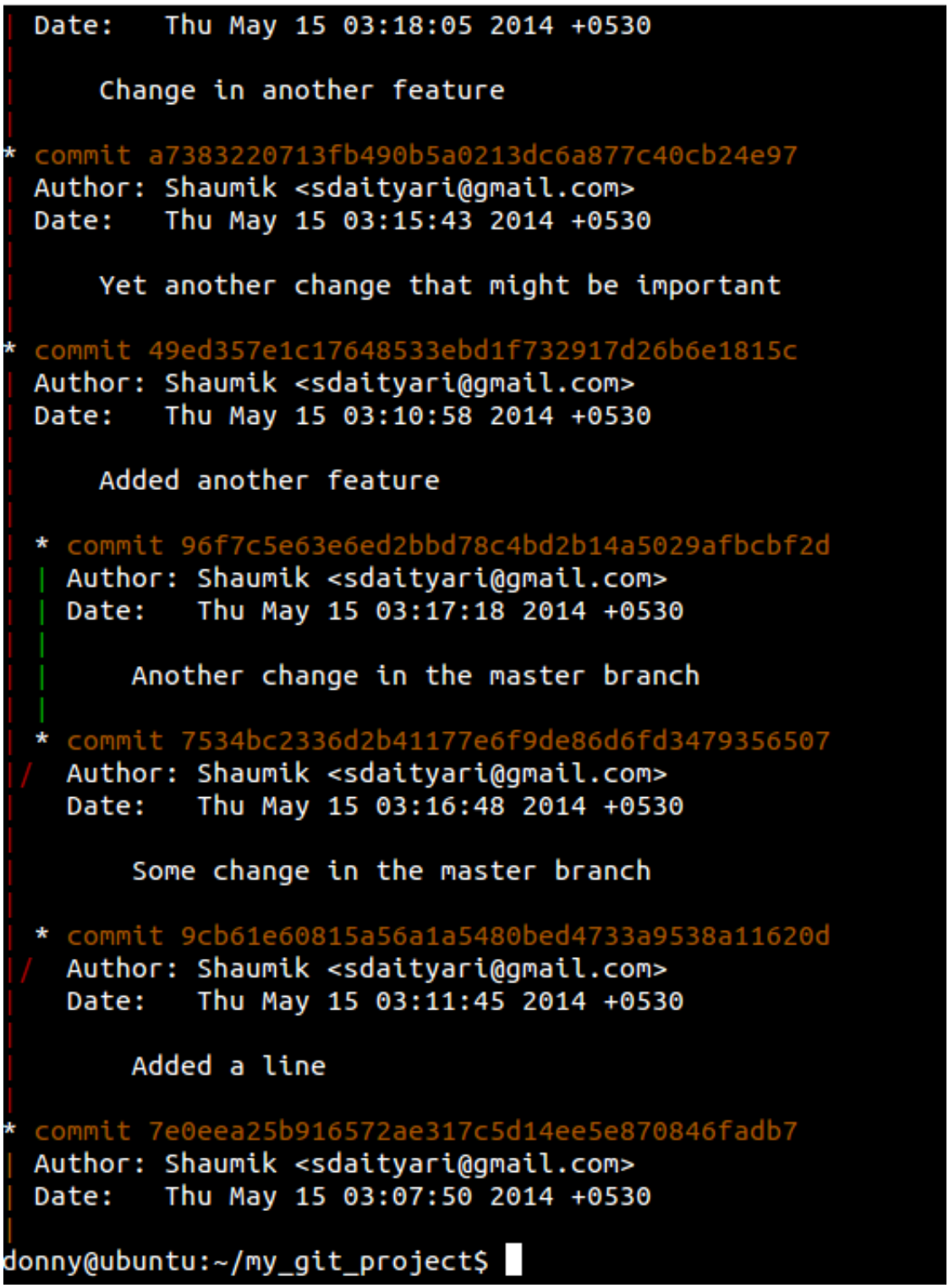
In an earlier post (http://www.sitepoint.com/using-git-open-source-projects/), I talked about the guidelines that one must follow while contributing to open source projects. Every new feature must be built on a new branch and a pull request or patch should be submitted from that new branch. The `master` branch must only be used to sync with the original source of the project.

There are a number of different models that are used by organizations to manage their projects. This post on git branching (http://nvie.com/posts/a-successful-git-branching-model/) discusses a generalized model that fits most organizations today.

Visualizing Progress Through Branches

Now that you know how to create branches, you must wonder how we could grasp the idea through a single command. GUI clients for Git do this with a single click, but you can do the same in the terminal. In our case, we have added a few commits to branches — `master`, `new_feature`, and `another_feature`.

```
git log --graph --all
```



Using `git log` shows the history of the project. `--graph` shows the direction of commits, whereas `--all` shows all branches. You may want to add `--oneline` to display commits in single lines (without their details, of course).

Merging Branches

Let us come to perhaps the most important part of this tutorial: The ability to merge branches in Git. One of the branch models we discussed was that all of the features are developed in separate branches and the ones with desirable results are merged

with the `master`. To do so, you need to `checkout` to the `master` branch and merge the `feature` branch with it.

```
git checkout master
git merge [branch_name]
```

```
donny@ubuntu:~/my_git_project$ git merge another_feature
Auto-merging my_file
CONFLICT (content): Merge conflict in my_file
Automatic merge failed; fix conflicts and then commit the result.
```

However, in our case, we see that a conflict has been raised. Why did that happen? There were changes made in the same file in both branches and Git wasn't able to determine which changes to keep, raising a conflict in the process. We will see how to resolve conflicts later in the tutorial.

Collaborating with Git

In the Git for Beginners (http://www.sitepoint.com/git-for-beginners/) post, we ended with a `git push` that synced our code on GitHub. Thus, you must know that to effectively send code from your local machine to another `remote`, you use the `git push` command. The different ways of doing a push are as follows.

```
git push
git push [remote]
git push [remote] [branch]
git push [remote] [local_branch]:[remote_branch]
```

The first command sends your code to the current branch of `origin` remote. When you specify a remote name, the code from your current active branch is pushed to the branch by the same name on the remote. Alternately, you can push from a local branch to a branch by a different name on the remote.

If you prefer the last command to push, there is a trick that deletes a branch on the remote. If you supply an empty `[local_branch]`, it will delete the `remote_branch` on the server. It looks like this:

```
git push [remote] :[remote_branch]
```

The `push` is not always accepted on the remote though. Take the following case.:

```
donny@ubuntu:~/my_git_project$ git push ssh_remote new_feature:another_feature
To git@github.com:sdaityari/my_git_project.git
 ! [rejected]        new_feature -> another_feature (non-fast-forward)
error: failed to push some refs to 'git@github.com:sdaityari/my_git_project.git'
hint: Updates were rejected because a pushed branch tip is behind its remote
hint: counterpart. Check out this branch and merge the remote changes
hint: (e.g. 'git pull') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
donny@ubuntu:~/my_git_project$
```

A ‘non fast forward’ branch refers to that fact that the commit at the tip of the remote branch does not match with any commits on my local system. This means that the remote branch has changed since the last time I synced it.

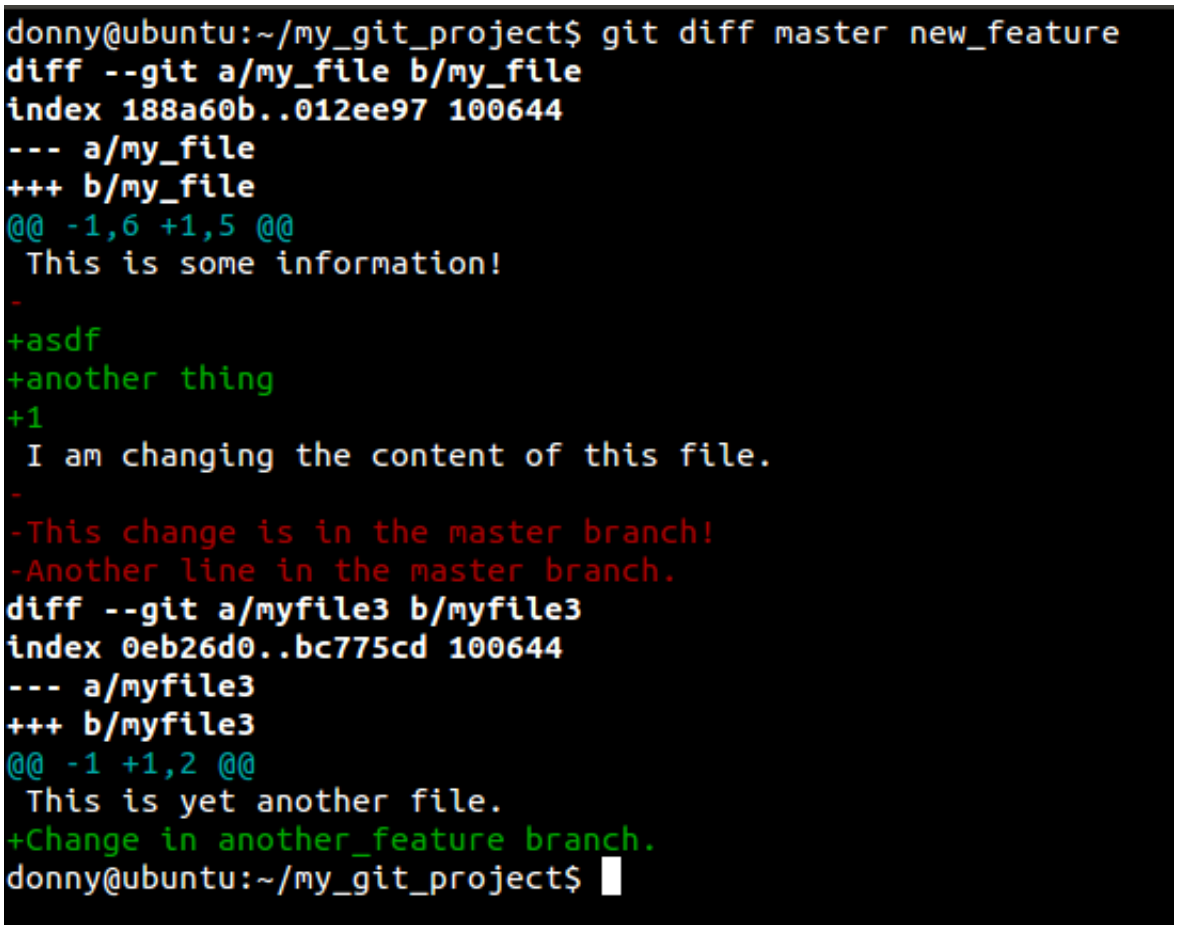
In such a situation, I would need to pull the changes from the remote branch, update my local branch and then push my changes. In general, it is considered good practice to always pull before a push. The syntax is exactly the same as the push counterparts.

```
git pull --rebase [remote] [branch]
```

Alternately, some people do a `git fetch` followed by a `git merge` rather than a `git pull`. Essentially, a pull does a fetch and then a merge (http://stackoverflow.com/questions/292357/whats-the-difference-between-git-pull-and-git-fetch). A `--rebase` should also be used, because it first pulls the changes to your branch, and then puts your work over it. This is desirable because all possible conflicts are raised based on changes that you made and hence you are in a good position to decide what to keep and what not to keep.

As another possibility, organizations that don't give every contributor `push` access to their repositories work through emails. In such a case, you would create a `diff` file by comparing the differences between either commits or branches. The `git diff` command shows the difference and the changes are stored in a file, which is then mailed to the organization. These diff files are popularly known as "patches".

```
git diff [branch1] [branch2] > [file_name]
```



The parts of a diff file that are generated are explained in detail in this post (http://www.sitepoint.com/understanding-version-control-diffs/). If you have a `diff` file and you want to apply the patch, you simply use the command below.

```
git apply [file_name]
```

Resolving Conflicts

Resolving conflicts is important in any Git workflow. It is natural that many people work on the same file and the server has a newer version of that file since the last time you pulled for changes. Conflicts can also arise when you try to merge two branches that have changes to the same file. Sometimes Git tries to apply certain algorithms to see if it can solve the conflict itself, but it often raises a red flag to make sure there is no loss of data.

When we tried to merge our branches, we noticed that a conflict arose. Let us see how we can resolve it. A `git status` shows you what caused the conflict.

```
donny@ubuntu:~/my_git_project$ git status
# On branch new_feature
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Changes to be committed:
#
#       modified:   myfile3
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      my_file
#
donny@ubuntu:~/my_git_project$
```

We can then open the identified file using a text editor (like VIM) to see what is wrong. The contents of the file are as shown:

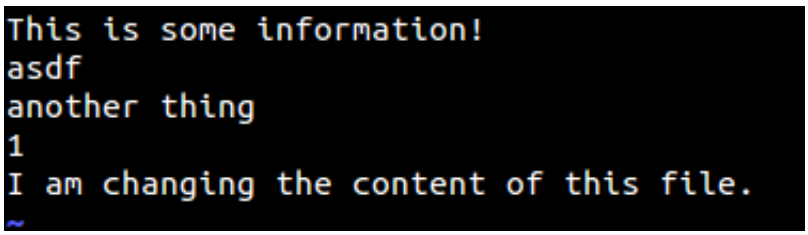
```
This is some information!
<<<<<<< HEAD
asdf
=======
another thing
1
>>>>>>> another_feature
I am changing the content of this file.
~
~
```

Look at the file carefully. Notice the three lines that have been inserted by Git.

```
<<<<<<<< HEAD
...
...
========
...
...
>>>>>>>> another_feature
```

What this means is pretty simple. The lines of code that are between `<<<<<<<< HEAD` and `========` are a part of your current branch, whereas those between `========` and `>>>>>>>> another_feature` are present in the branch that you are trying to merge. What you need to do is remove those three marker lines and edit the content in between them to something that you desire. In our case, let's keep all the information, so the end file will look something like this.

Although we keep all the content within both blocks, you could remove all content, keep some of the content, or write something else entirely in their place while resolving the conflict.



This conflict was pretty simple in our case. In more complex projects, you might have a lot of instances that lead to conflicts, with a number of instances of the lines shown above included in the conflicting files. The process of solving a rather complex conflict remains the same: Check the affected files through a `git status`, open each of them, and search for occurrences of `HEAD`.

Conflicts can arise while pulling from a remote too. In that case, the last line would read `>>>>>>>> [commit_hash]` instead of `>>>>>>>> [branch_name]`, where `[commit_hash]` would be the identifying hash for the commit.

Conclusion

Git is popular in the developer community, yet critics often compare it to a Swiss Army knife. It is true that Git offers a lot of features. But once you understand the basics, they are very intuitive to use. Let's hope this tutorial can get you started with Git in a way that allows you to contribute in a team environment.

If you have anything to add on any of the features discussed above, please add your comments.

Tags: gihub (http://www.sitepoint.com/tag/gihub/), git (http://www.sitepoint.com/tag/git/), git branching (http://www.sitepoint.com/tag/git-branching/), git merging (http://www.sitepoint.com/tag/git-merging/)

Shaumik Daityari (http://www.sitepoint.com/author/sdaityari/)

(https://twitter.com/ds_mik) (https://plus.google.com/+ShaumikDaityari) (https://www.facebook.com/shaumikdaityari) (https://github.com/sdaityari/)

Shaumik is an optimist, but one who carries an umbrella. An undergrad at Indian Institute of Technology Roorkee and the co-founder of The Blog Bowl, he loves writing, when he's not busy keeping the blue flag flying high.

windows desktop using Eclipse,
I have installed GIT plugin in eclipse Private and public key is generated and public key is copied to /root/.ssh/authorized_keys (I used root to install GIT on linux machine)

Host name : 192.168.116.132
user root
repository : project.git

Protocol : ssh

showing network error while connecting.

Shaumik Daityari ➚ sanjeev kumar • 7 months ago

It is possible that your network is blocking the connection. Are you sure the SSH port (22) / Git over the SSH port is open on your network?

Another issue could be that your VM doesn't accept connections over the SSH port. Go to the settings (in your provider's website) and add the SSH port in case it's not accepted. This process is different from provider to provider.

sanjeev kumar ➚ Shaumik Daityari • 7 months ago

Thanks for you quick reply,

SSH port is open and I am using it to connect to tat VM,
Is there any pre-requisite before setting it

Shaumik Daityari ➚ sanjeev kumar • 7 months ago

Okay, is the SSH port set to the default 22? Or do you use ssh username@xyz.com:[port] to connect?

Also, which VM do you use?

sanjeev kumar ➚ Shaumik Daityari • 7 months ago

All tutorial which I have checked is for GITHub not any one for GIT,
I am able to connect git from Sourcetree of same repository, but failing with Eclipse.

Shaumik Daityari ➚ sanjeev kumar • 7 months ago

I haven't used Eclipse much, but I think it could be some issue with the configuration (since SourceTree is connecting fine).

sanjeev kumar ↗ Shaumik Daityari • 7 months ago

Thanks Shaumik,

I am able to connect now, there was some issue with key setup.
Please send me some detailed configuration from Eclipse to GIT if you have.

 • Reply • Share ›

Shaumik Daityari ↗ sanjeev kumar • 7 months ago

Hi Sanjeev,

Sorry, but I don't use eclipse :)

 • Reply • Share ›

sanjeev kumar ↗ Shaumik Daityari • 7 months ago

yes its default port 22 and I am able to login on my linux (CentOS) through putty from my windows machine, Linux is installed on VMware workstatio 9

 • Reply • Share ›

Pulkit seth • a year ago

Hi, Very nice article but i have couple of queries that i would like to discuss here, the very first is when we are creating the branches using this command :

git branch [branch_name] where is this branch created ? I cannot see this on github, still there is master branch only.

Thanks

 • Reply • Share ›

Shaumik Daityari ↗ Pulkit seth • a year ago

The 'git branch' command creates a branch only in your local repository. You need to push the branch to GitHub to see it there.

git push origin branch_name

 • Reply • Share ›

Pulkit seth ↗ Shaumik Daityari • a year ago

Thanks for reply, actually i am a beginner. So initially i have to push my changes to my local branch and then to push it on the master branch ?

I would like to know the clear flow in steps, right now i am cloning the master branch into my localhost then i will push my changes to my local branch and then i have to merge it with the master branch ? Is this the correct way to work with github ?

 • Reply • Share ›

Shaumik Daityari ➚ Pulkit seth • a year ago

Hi Pulkit,

If you make a commit, your changes are saved in your local repository.

If it's a personal project, you commit (which gets saved in the local repo) and then push to a desired branch GitHub. For the repositories of others, there's this concept of forking, which I have explained here- www.sitepoint.com/using-git-op...

I'd suggest you go through this first if you haven't already :) www.sitepoint.com/git-for-begi...

 • Reply • Share ›

Bhaskar Chowdhury • 2 years ago

I haven't seen you mentioned about "tig"...I believe that tool would useful for people want a better viewing.

 • Reply • Share ›

Michał Paluchowski • 2 years ago

I completely agree that rebasing when pulling is better than the default merge performed by git pull - leaves a much cleaner history. However, rebasing is not simple :) Doing it right requires one to understand well what a rebase does and how it affects history. I had to repair more than one local repository (and a few remote ones) where people tried rebases and ended up with odd, disconnected histories.

Also, if I'm about to push a merge commit to a remote, while the remote has some commits I need to pull first, then I'll need to do a rebase --preserve-merges, otherwise my merge commit will get "unpacked" into single commits, which further borks history.

 • Reply • Share ›

Shaumik Daityari ➚ Michał Paluchowski • 2 years ago

Hi Michal

COURSES >
(https://www.sitepoint.com/premi
q=&content_types[]=Course&utm

1:06:23

Faster Websites with Nginx

Chris Lea

★★★★½

(https://www.sitepoint.com/premium/cou
websites-with-nginx-2757/?
utm_source=sitepoint&utm_medium=rela

1:17:14

Build a Corporate Website with Joomla

Kray Mitchell

★★★★☆

(https://www.sitepoint.com/premium/cou
a-corporate-website-with-joomla-
2726/?
utm_source=sitepoint&utm_medium=rela

3:07:36

JavaScript: Next Steps

M. David Green

★★★★★

(https://www.sitepoint.com/premium/cou
next-steps-2921/?
utm_source=sitepoint&utm_medium=rela

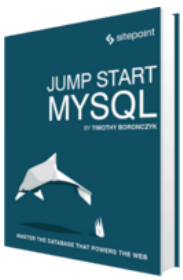
BOOKS >
(https://www.sitepoint.com/premi
q=&content_types[]=Book&utm_s

Level Up Your Web Apps With Go

Mal Curtis

★★★★½

(https://www.sitepoint.com/premium/boo
up-your-web-apps-with-go/?
utm_source=sitepoint&utm_medium=rela



Jump Start MySQL

Timothy Boronczyk

★★★★½

(https://www.sitepoint.com/premium/boo
start-mysql/?
utm_source=sitepoint&utm_medium=rela

Deliver First-Class Websites: 101 Essential Checklists

Shirley Kaiser

★★★★☆

(https://www.sitepoint.com/premium/boo
first-class-websites-101-essential-
checklists/?
utm_source=sitepoint&utm_medium=rela

SCREENCASTS >
(https://www.sitepoint.com/premi
q=&content_types[]=ScreenCast&

Working with a Database on the Command Line

Dr. Richard Stibbard

(https://www.sitepoint.com/premium/tuto
with-a-database-on-the-command-
line/?
utm_source=sitepoint&utm_medium=rela

Creating MySQL Databases and Tables From the CLI

Dr. Richard Stibbard

(https://www.sitepoint.com/premium/tuto
mysql-databases-and-tables-from-
the-cli/?
utm_source=sitepoint&utm_medium=rela

MySQL on the Command Line

Dr. Richard Stibbard

(https://www.sitepoint.com/premium/tuto
on-the-command-line/?
utm_source=sitepoint&utm_medium=rela

Visit

SitePoint Home (/)
Forums (https://www.sitepoint.com/community/)
Newsletters (/newsletter/)
Premium (/premium/)
References (/sass-reference/)
Shop (https://shop.sitepoint.com)
Versioning (https://www.sitepoint.com/versioning/)

Connect

(http://www.sitepoint.com/feed/) (/newsletter/) (https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (https://plus.google.com/+sitepoint)