AUG 15, 2012

EXPLAINING THE BASIC CONCEPTS OF GIT AND HOW TO USE GITHUB

There are a lot of great projects hosting their code at github, and Thematic has moved there too. Github has a great help section with articles and tutorial of how to get started, but it can seem daunting if you are not comfortable with the command line.

Today, there are several GUIs around for working with git so the command line is not necessary for the basic workflow. But you still need to understand what you are doing. This is a guide for those of you just trying to get your head around git and how github works, without a single line of code.

I am assuming you are familiar with some kind of coding: at least html and css, and perhaps even some php. But you are a beginner with source control and want to begin using github. I will walk through the concepts and workflow, while introducing some best practices and tips along the way that will hopefully make your life easier. I will use Thematic as my example here, but the process works the same with any github project.

THE GITHUB WORKFLOW

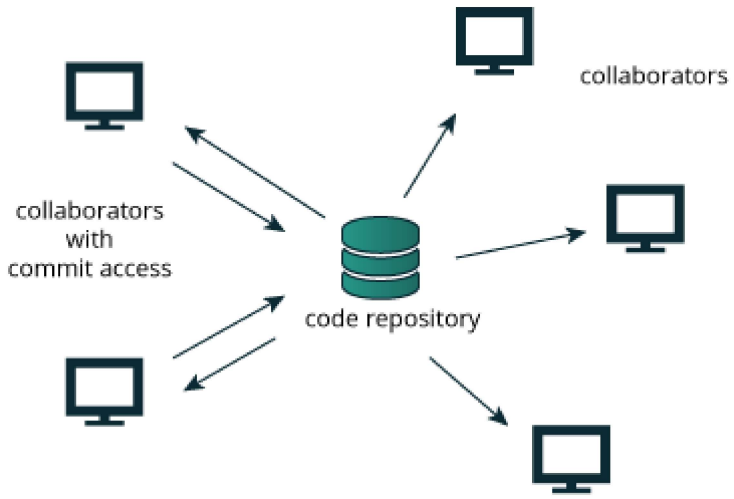
This is the workflow I recommend you use when you want to contribute to projects on github.

1. **fork** a project on github.
2. **clone** your github fork to your computer
3. **create a topic branch** for your own work in your local clone
4. **commit** changes to your local repository
5. **push** the changes to your github fork
6. send a **pull request** back to the original project
7. get a **warm fuzzy feeling** inside for contributing to open source

These are the basic things that you need to understand, and I will go through all of the terminology above and a little bit more. I will not show you *how* to do it, but rather explain *what* these things mean, so that afterwards you can use which ever tool makes the most sense to you.

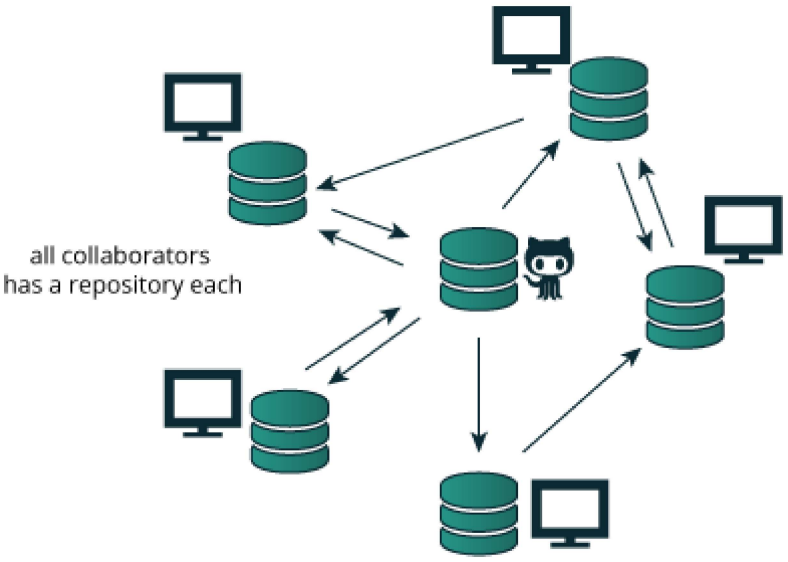
CENTRALIZED VS DISTRIBUTED VERSION CONTROL

Git is a distributed version control system, as a contrast to for example subversion that is centralized. What this means is that with subversion, the code repository (repo) lives on a server and people need special permissions (commit rights) to be able to add code to the repo. Everything lives in one place.



Centralized version control

In a distributed system, you get a repository of your own when you clone the project. This means you can work and add code to the repo even when offline, since the repository lives on your computer. But this also means you need a bit of discipline in keeping in sync with the rest of development, since your repository is separate from the others.

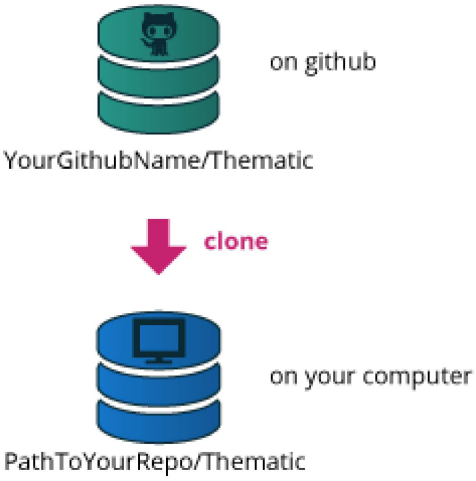


Distributed version control

FORK AND CLONE

First of all, we need to fork. In github, all you need to do is click that fork button and you get your own personal repository of Thematic copied to your github user account.

When you want to actually work on your repository you need to `clone` it to your computer, and now you own two repositories: one remote on github and one local on your computer.



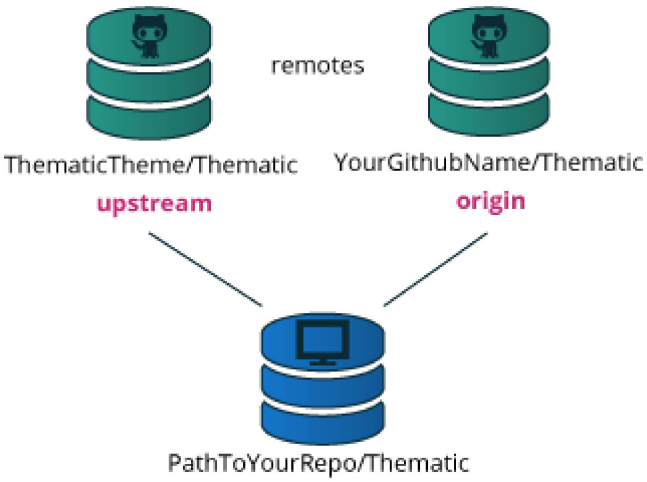
It's called forking on github because every repository is a potential for taking development in your own direction. The choice to contribute back to the original project is yours to make, but of course you want to do that — that's why you are reading this article, right?

A LITTLE THING ABOUT REMOTES

Git calls other repositories that it knows about remotes. By default, when you clone a repository it will save the parent repository under the name `origin`, so origin on your computer repository will be pointing to your personal Thematic fork on github.

You can add as many remotes as you like and call them whatever you like. You could add a remote for everyone else that has also forked Thematic if you want to keep track of their work.

One thing you need to do if you want to keep up with the main development is to add the Thematic github repository as a remote, and the naming convention for this remote is `upstream`.



This means you will have two remotes configured in your local repo, `origin` would point to your fork on github and `upstream` would point to the main Thematic github repo. You can read a bit more on the github forking help page.

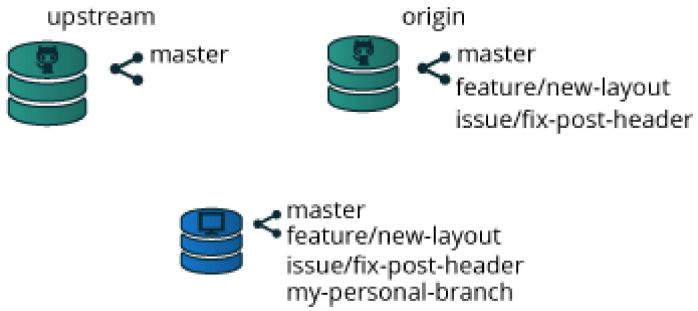
CREATE A TOPIC BRANCH

Git repositories are organized with tags and branches. A branch is a way to keep lines of development separate. The default branch in git is usually named master. Branches made off of master is commonly referred to as topic branches.

An example use case for branching is when you are working on a website and have an idea but working on it might break something. Create a branch for your idea, commit your work there and do your testing, and when everything works as intended you `merge` the branch back into master.

Another use case could be that you have two ideas you want to compare, like two color schemes for your website. Make a branch for each idea and you can easily switch between them for comparison.

When you create a branch, it will exist only on your local repo unless or until you decide to share it with others. And other repositories might have several branches that you choose to copy and track or decide to leave alone. You can have as many branches as you like.



A common thing for software development is to make a topic branch for each bug ticket that gets worked on. When a bug has been resolved, that branch is `merged` into master. That way, work can be done one several bugs in parallell without interfering with each other.

Big projects might have specific workflows that developers are expected to follow – naming conventions and such. Personally, I'm a fan of git-flow workflow proposed by Vincent Driessen. You certainly don't need to do anything as complicated as that, but it's a very good idea to make a topic branch for your own work when working on github forks. I will get into more detail regarding that when we come to sending pull requests.

TAGGING

Tags are pointers to a certain commit, and just an easier way to reference them than memorizing hash numbers. A common way to use tags is for version numbering. Github vill generate zip files ready for download using your tags, Thematic's can be found at https://github.com/ThematicTheme/Thematic/tags. But they are not needed for this workflow, see them as an optional extra at this point. I just wanted to include a note about them for the sake of completeness.

COMMIT YOUR WORK AND CHECKOUT THE FILES

When you have made some changes that you want to save, you `commit` them to your repository. Commits are logical chunks of changes that gets saved in sequence, forming a history that you later can go through. Changes to several files can go in the same commit, it's like you are saving the state of all the tracked files in

your directory at once. Each commit gets a timestamp, an author name and email (which comes from a configuration file), a commit message and an automatically generated hash.

The hash is how git keeps track of its commits and references them. A typical hash looks like `27b6b79fca466af4648f9f8042e1f159b392293d`, but fortunately git is smart enough that you often only need to give the first six or eight characters for it to know which commit you are talking about. Github lets you browse the commit history and the hash of each commit is shown to the right in the list.

THE COMMIT MESSAGE

If the hash is how the computers keep track of the commits, the commit message is how people keep track of them. Do take your time to write good commit messages. Compare a commit log of

- some css
- styling
- ooops
- misc fixes and cleanups

versus

- add subtle background pattern to body
- make subheadings larger on archive pages
- fix typo in site footer
- cleanup code with htmltidy

and I think you know which one is preferable in the long run. Since commits can be moved around between branches and repositories it is a good idea to be descriptive. Keep messages in the active, present tense. Think of the commit message as an answer the question “What will this commit do if I add it to my repo?”.

As a github feature, if you include a github issue number in your commit message (“#xxx”), it will be attached to that issue in the issue tracker. And if you use any keyword of `fix #xxx`, `fixes #xxx`, `fixed #xxx`, `close #xxx`, `closes #xxx`, `closed #xxx`, that issue will get closed automatically.

ALTERING THE HISTORY

Commits are not set in stone. They can be manipulated afterwards, split apart, combined into one (squashed), the message can be altered (amended) etc etc. Keep in mind though that the hash is individual to a commit and if commits are changed then the corresponding hash changes as well. This will alter the history that git keeps track of and should not be done on any branch that has been shared with others.

One advantage with using local topic branches is that you can manipulate commits in them without worrying about messing up other people's history. Once you have shared a branch though, leave the commits alone.

USING CHECKOUT

If you want to see how the files of your project looked at any point, you can `checkout` them. You can use the hash-tags of a commit as a reference, but most common is to use branch names or tags.

This is how you switch between the branches, you `checkout` one branch and then `checkout` the other. Git has a special kind of tag called `HEAD` that keeps track of which commit is currently checked out in the active directory. So you can checkout how the files looked five commits from now, and then revert back to the latest version by checking out your latest commit (which is usually a branch name). Think of it as undo/redo on steroids.

You see, git doesn't save multiple copies of the files themselves, but only the actual changes from one point to the other. The repositories on github contain only the commit history and not any actual files. This is why repositories are so small compared to keeping several backups of actual files. Only the changes are recorded and saved. When you checkout something, those changes get applied to the files in your directory.

PULL = FETCH AND MERGE

Ok, now the files on your computer are in a repository of their own, locally on your computer. So how do you keep in sync with the

other repositories, the remotes?

This is done with fetching and merging. You `fetch` information from `upstream` and `merge` those changes into your own repo. `Fetch` will only connect to the remote repository and download the latest commits to your history, `merge` will actually put those commits into one of your branches and, um, merge them with your own commits. The history is not specific to one branch only, it keeps track of all branches at the same time.

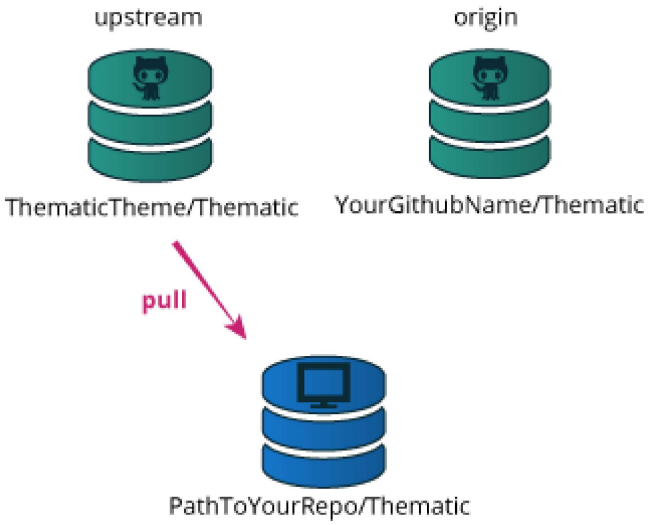
THE MERGE

Merging simply means git will try to combine changes from two places, by placing commits in a row after one another. You can think of it as a zipper putting commits from two sides together in one line, although it's not one from every alternating side like the zipper does but from oldest to newest using the timestamp.

If two commits try to change the same piece of code, you get what is called a merge conflict and you will have to manually look at the files and determine which version you want to keep. But most of the time there are no conflicts and the merge happens without problems.

THE PULL

It can be tedious to always fetch changes first and then merge them, especially when you are certain you want to put the commits into your branch. Fortunately, there is a command that will do both things at once, `pull` .



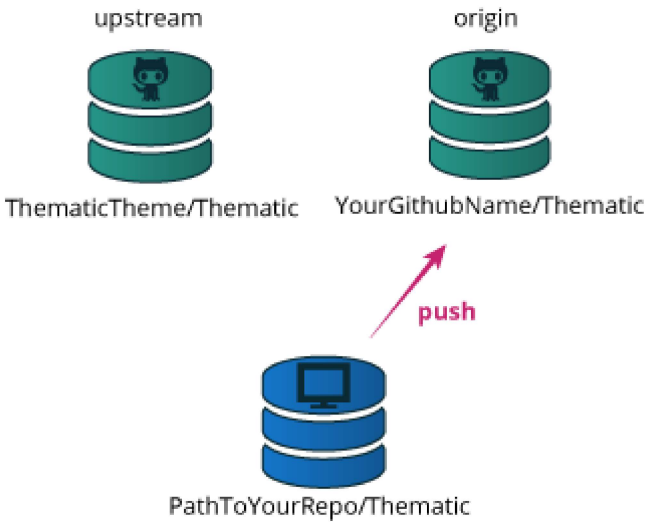
You can pull from any repository you like, you don't need special permissions to do a pull. Usually you need specify which repository and which branch you want to pull from, and the pull will happen to the branch that is currently checked out in your directory. You can set up automatic tracking of branches but that is a bit out of scope for this walkthrough.

Here comes the advantage of using topic branches. You can keep your local master branch clean so it will always be a mirror copy of the upstream repo, and your own code lives in topic branches.

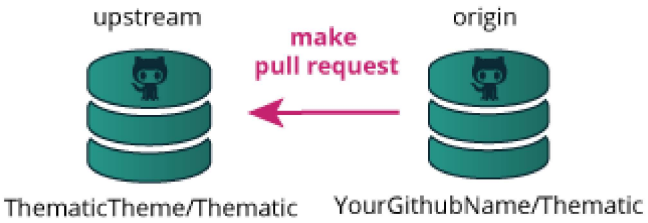
When upstream changes, you can directly pull to your local master branch without worrying about code conflicts. You can then update your topic branch if needed by merging your local master branch into the topic branch. Pulling and merging is also possible between local branches, and that will bring the latest changes over from master to the topic branch.

PUSH, OR SEND PULL REQUEST

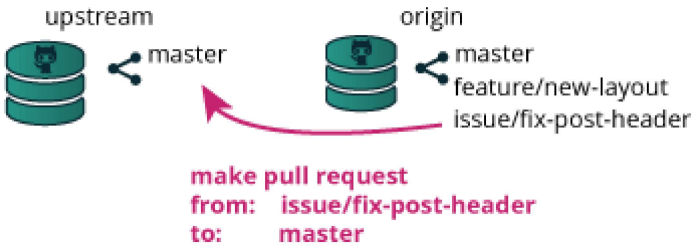
When you commit to your local repository, the commits only exists on your machine. If you want to share your code with others, you can `push` it to repositories where you have access. Your own github fork called `origin` is of course one. Push to origin and your commits will now be on github! Yay!



If you want to contribute to places where you don't have push access, like the Thematic theme repo, you can send them a pull request on github. You can't push changes to them, but they can pull changes from you.



Again, here is an advantage with the topic branch approach. Github lets you specify which of your branches you want the pull to target (the head), as well as which upstream branch you want the pull to merge into (the base). But there can only be one active pull request per branch. If you have more than one thing you want to contribute with, then you need to have a separate branch per contribution.



Once your pull request is accepted and merged into upstream by the project maintainer, then you can delete the topic branch, both on your computer and on your github repo (the “origin”). Your commits will now be part of the main project and the next time you do a pull from upstream, they will be in your master branch.

KEEP UP WITH THE DEVELOPMENT

Continuing development on the project means you will need to keep up with the upstream and make sure that it’s all kept in sync. This is where the discipline part comes in. Git doesn’t give you any automatic notification of changes upstream, although github’s follow feature will help you to keep track of what’s going on. But you still need to download the changes to your local repo manually.

So your new workflow will be:

1. **pull** the latest changes from upstream to your local master branch
2. **push** your master branch to your github repo so it is kept in sync
3. **create a local topic branch** from your local master branch when you want to do some work

4. **commit** your changes to your local topic branch
5. **push** the topic branch to your github fork
6. send a **pull request** to the original project from that topic branch
7. get your pull request accepted and do a small **dance of joy**

Phew! Now that you have managed all the way through this article I hope that you have a bit clearer picture of the steps involved. If you want to see a bit of commands that go along with these steps then read this guide on How to properly contribute to open source projects on GitHub. Go grab yourself a beer to celebrate your new understanding and come over to github and join the fun!

In Uncategorized
#beginner #git #tutorial

‹ PREVIOUS POST NEXT POST ›