Git for Beginners

By Ryan Florence, published 2010-09-12

Part of the issue Git Your Act and Deployment Together.

Every current or aspiring developer, member of a team or work-from-home freelancer, ought to be using source control for their code. It allows you to make mistakes and be experimental without worrying about getting back to an application that once worked. Working from multiple machines was a drag and collaborating with others was nearly impossible before the bearded computer geniuses came up with stuff like Git, my source control program of choice.

This article is all about Git, and using git from the command line. There are some full-featured GUI's for git, but most developers find the command line coupled with a GUI viewer to be more efficient. I know, Linus has no beard, but he should.

Basic Workflow

If you haven't already installed git on your machine, you should read the Installing Git article in this issue.

Starting with the basics here, we'll create a "repository", make some changes, and "commit" those changes into the repository. The point is to have a history of your code that you can always go back to. First open up the terminal, create a directory, and navigate into it:

```
1.  $ cd ~/Desktop
2.  $ mkdir my_app
3.  $ cd my_app
```

Step 1: Create the repository:

```
1.  $ git init
2.  Initialized empty Git repository in ~/Desktop/my_app/.git/
```

Git created a repository in `.git`. If you were to open the hidden directory you'd see something like this:

```
my_app/
  .git/
    config
    description
    HEAD
    hooks/
    info/
    objects/
    refs/
```

Unlike SVN, this is the only place you'll find any git folders, it doesn't make a `.svn` in every single sub-directory (yay!) There's nothing really to do with this directory right now, just know that it's there. If you get all uncomfortable just delete `.git` and all of the git stuff is gone. It's also good to know if you're deploying old-school-unenlightened-ftp-style, you might want to omit the `.git` directory in your upload.

Step 2: Do stuff. Create a README file with your text editor and save the file into `my_app`, or go command-line commando like so:

```
$ echo "I am a git repo." > README
```

To see what's going on run the `status` command. It will show you which files aren't being tracked, have been modified, deleted, etc.

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
```

Step 3: "Stage" your changes.

```
$ git add README
```

Staging your changes sometimes feels like an unnecessary step to beginners and svn converts. In essence we're just telling git that we want README to be

included in the next "commit." You can pass in a file name or any path, I usually just do

```
$ git add .
```

and catch everything. You also have to add modified files that already exist if you want them in the next commit.

Step 4: Commit the changes

```
$ git commit -m 'This is my first commit evar'
[master (root-commit) 206e419] This is my first commit evar
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Anything that was "staged", or added, gets committed to your repository. A "commit" is sort of like a snapshot of your code. Almost like you make a zip file and store it somewhere (except way more efficient.) It will always be there; you can refer to it whenever you want.

Now we wash, rinse, repeat: make changes, add, commit.

```
$ git add .
$ git commit -m 'whatever message you want'
```

If you have only modified files, and haven't created any new ones, you can use this shortcut and skip the `git add` step, but remember this will only catch files that have *already been added before*:

```
$ git commit -a -m 'whatever message you want'
```

Step 5: Revert some changes

Make some changes to `README` in your repository, then run this command:

```
$ git checkout README
```

Back in your text editor the file should be back to how it was the last time you made a commit.

These four simple commands, `init`, `add`, `commit`, and `checkout` will enable you to

create a history of your code and remove all sorts of risk in your development workflow.

Remote repositories (on github)

Working with a remote repository is a great way to work from multiple machines, collaborate with other developers, and back up your source code. A remote repository is simply another repository, typically on another server, that you can push your commits to and pull other commits from.

If you don't already have an account on github, go get one, because we'll be using it in the next examples.

Step 1: Create the repository on github. Log in and click "New Repository." Name it `my_app` if you'd like.

Step 2: Add the remote repository to your local "working tree." Open up the terminal and navigate back to `my_app` (perhaps it's still open?) We're going to add a "remote."

```
$ git remote add origin git@github.com:your_name/my_app.git
```

The syntax is `git remote add <name> <url>`. We named this remote "origin" out of convention. You can have multiple remotes and give them all different names like 'stage' or 'production'. Check out all your remotes with `git remote` or `git remote -v`.

Step 3: Push your commits to the remote repository.

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 242 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /example/my_app.git
 * [new branch]      master -> master
```

Syntax is `git push <remote> <branch>`. We haven't talked about branches yet, but master is the default. You can see it packages everything up and then sends it to the remote repository. If you've had several commits, it pushes them all up. I encourage all of our developers to commit and push often.

Step 4: Pull changes from the origin repository. It's good practice to pull before you push, ensuring you have the latest code. In fact, I pull just for the heck of it throughout the day.

```
1.  $ git pull origin master
```

Same syntax as push, `git pull <remote> <branch>`. This fetches any commits the remote has that your local working tree does not, and automatically merges them in. Typically the merge happens without any problems, but occasionally you'll have conflicts which are usually straightforward to fix.

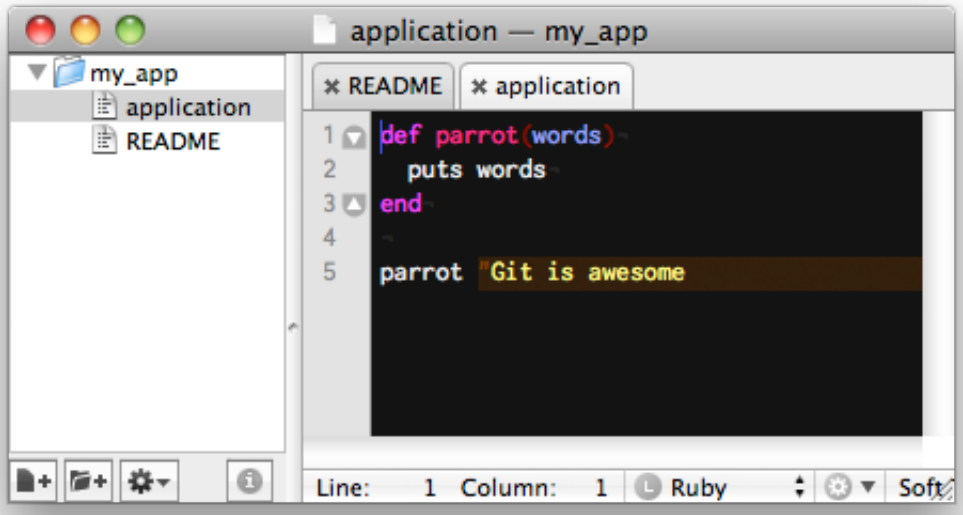
Advanced Workflow

I hesitate to call this "advanced" since git is so powerful, but for a beginner, this is advanced :P

Generally you don't want to work on the "master" branch–instead you create topic branches of code, work, and then merge branches back together. Git is flexible so you can create whatever branching / workflow model you want, I'll present a very basic one here (yes, a basic, advanced, workflow).

While developing new features you can't get away from fixing bugs in the current release. Let's explore a scenario where we start developing a new feature and fix bugs on the current release at the same time.

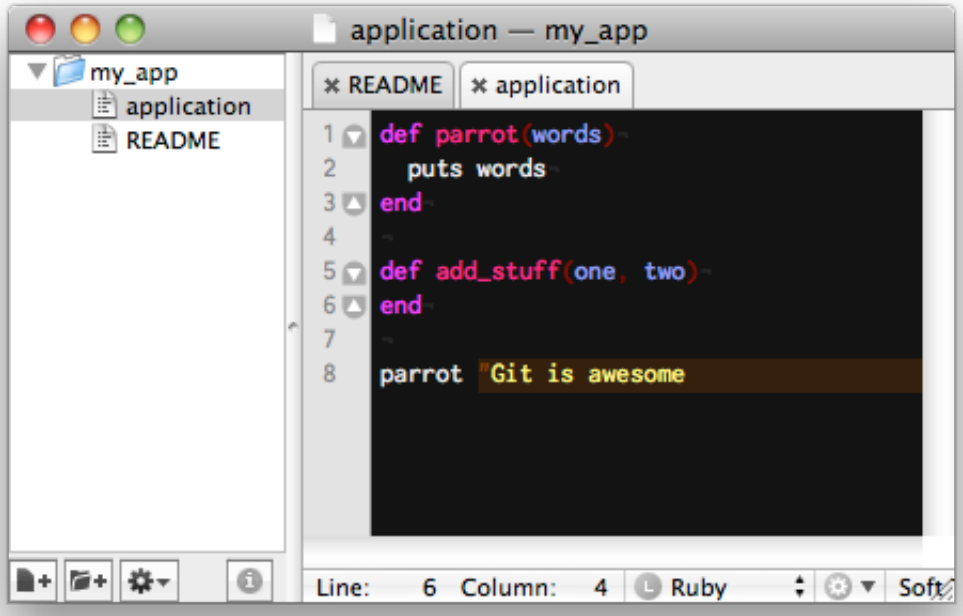
Here's a snapshot of my app right now.



Check out a new feature branch: First, check out a new branch in which to develop the feature. After that's created make some changes and commit them.

```
1.  git checkout -b develop-awesome-feature
```

Make some changes



Now commit.

```
git commit -a -m 'started the add_stuff method'
```

Oops! We're missing a closing quote on line 8. This application is in production so we need to fix this quick, can't wait to finish `add_stuff`.

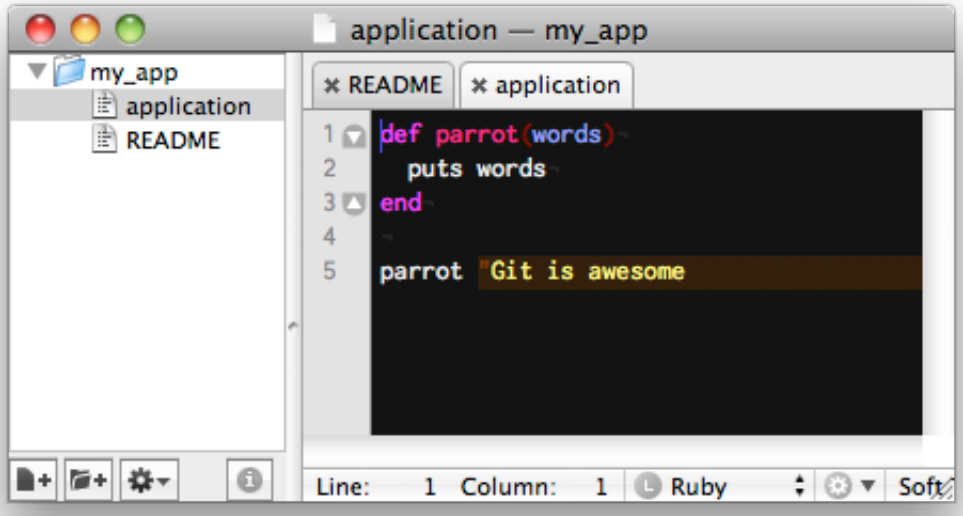
Check out a bug fix branch: We need a new bug fix branch to work from. Before we do that, make sure you know which branch you're on:

```
$ git branch
* develop-awesome-feature
  master
```

That asterisk tells you where you're currently working. Since our bug fix is high-priority, and will probably be finished before the new feature, we first switch back to the master branch and then branch off from there. Note that last time we did `git checkout -b`, that's the syntax for creating new branches, once created you simply use `git checkout`.

```
$ git checkout master
```

Look at the file now:



All of our new feature code is gone, which is exactly what we want. Now we can fix the bug in its own branch and update the application without a bunch of unused–and potentially broken–development code from our feature; nor do we have to wait for `add_stuff` to be completed before we can release the fix.

Liberating? Yes. Okay, back to work. Check out a new branch for the bug, add a quote to line 8, and commit:

```
$ git checkout -b develop-bug-quotes
Switched to a new branch 'develop-bug-quotes'
```

Now add a quote to line 8 and commit.

```
$ git commit -a -m 'fixed quote bug'
[develop-bug-quotes 89d6037] fixed quote bug
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Merge the bug fix into master: Since we're working on our bug fix branch, the master branch doesn't have the fix yet. We need to merge the bug fix branch into master. Merging like this is cake, but first you have to pay attention to which branch you've got checked out. When you merge git will merge a branch into the branch you're working in, so let's checkout master first.

```
$ git checkout master
Switched to branch 'master'

$ git merge develop-bug-quotes
Updating 48d01de..89d6037
Fast-forward
 application |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

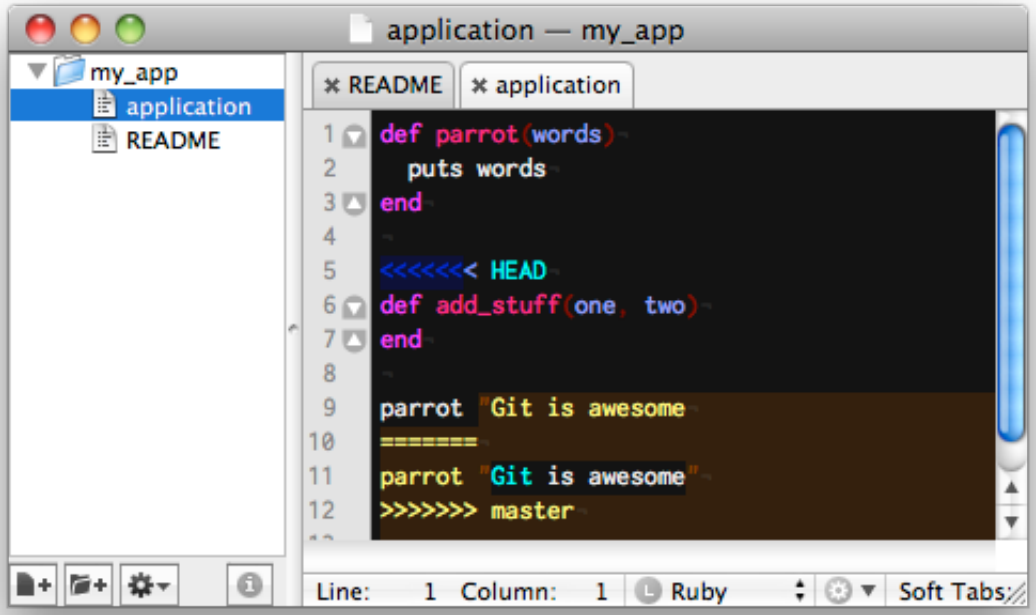
Now you could deploy this application without waiting on the new feature to be complete. Pretty slick. Before we move on, you really don't need that branch hanging around forever. Let's delete it–don't get all nervous, git will complain if your changes aren't merged yet and won't delete the branch. (You'll find it's nearly impossible to lose anything when you're using git.)

```
$ git branch -d develop-bug-quotes
```

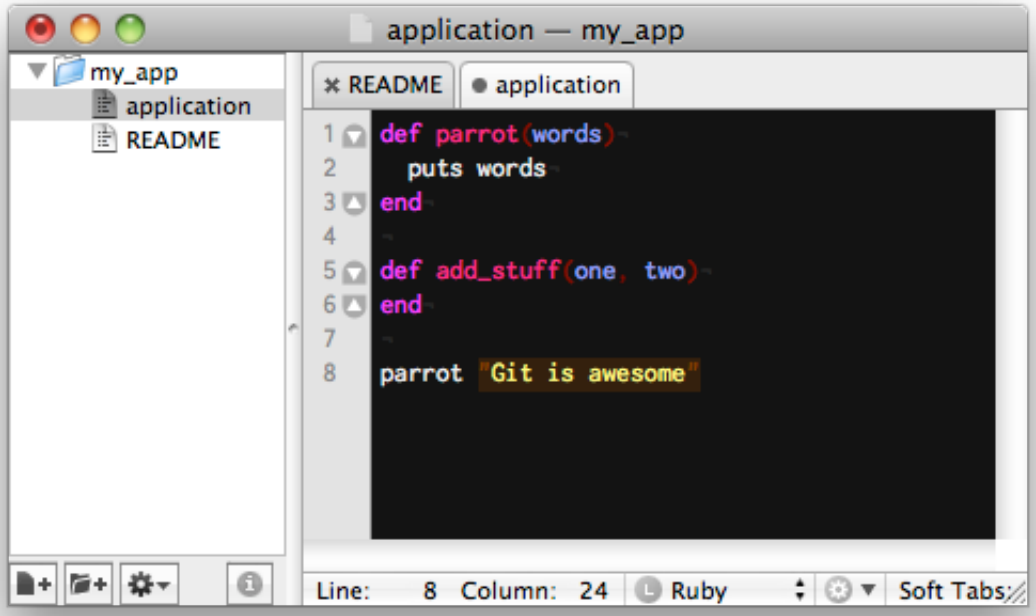
Merge the bug fix into the feature branch: This is the part you've been waiting for. Switch back to the `develop-awesome-feature` branch and look at the code, we've got the bug there again! Let's merge once again.

```
$ git checkout develop-awesome-feature
$ git merge master
Auto-merging application
CONFLICT (content): Merge conflict in application
Automatic merge failed; fix conflicts and then commit the result.
```

Ah snap! A conflict. Usually my merges don't have any conflicts, especially when I'm working on my own–but that's not to say I don't deal with conflicts almost daily. There are some great merge tools to help you here. I'm on a Mac and really like using the built in FileMerge application for conflict resolution. I'll show how to use that, but first let's take a look at doing it with any ol' text editor. Git added some stuff to the file, take a look:



Line 5 shows what's in the current "HEAD", (your checked out branch, basically) and then after the `======` it shows what's in `master`, or what git tried to merge. Simply get rid of all the garbage and make sure things look the way you want them to:



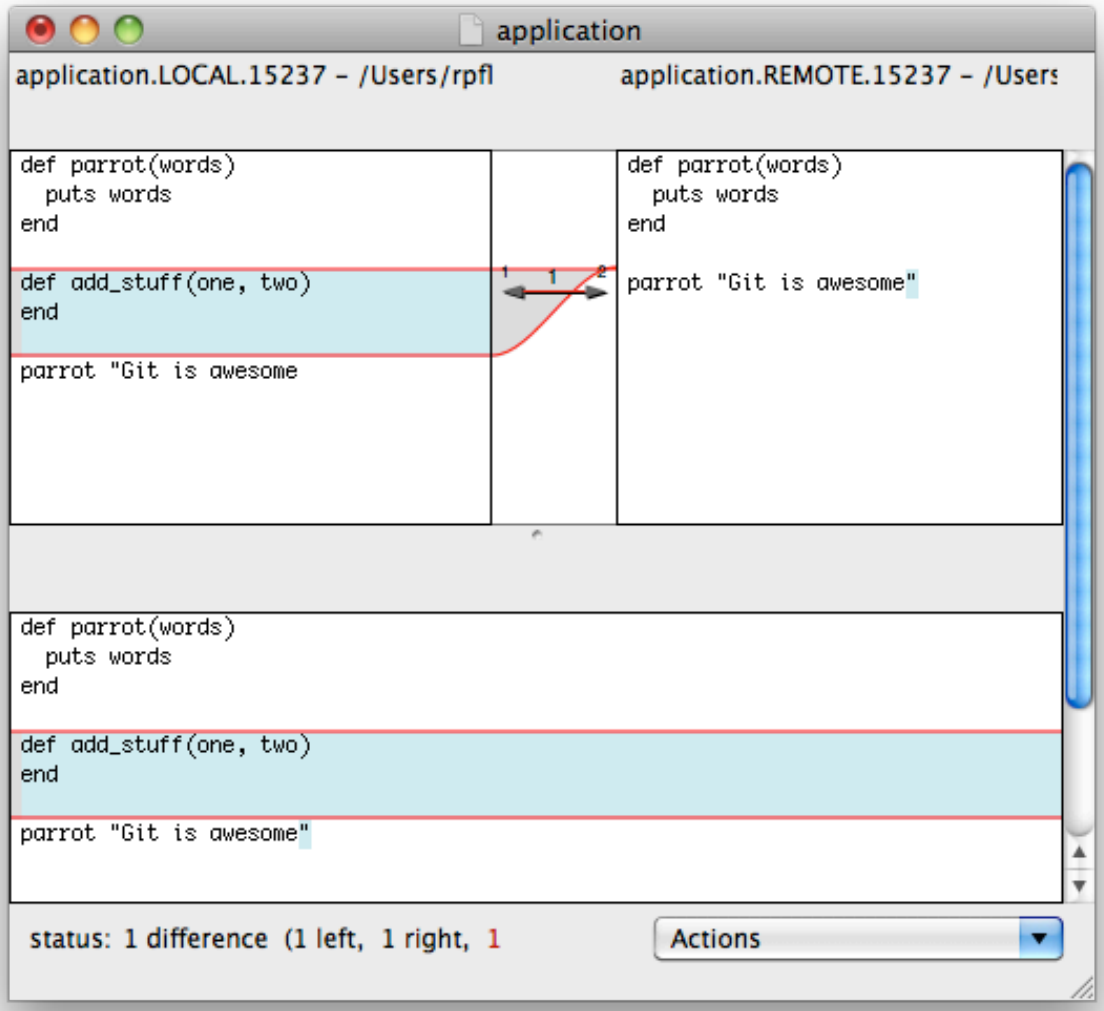
Save the file, and commit.

```
$ git commit -a -m 'merged with branch master'
```

Let's take a look at how to use `mergetool` the next time you have a conflict.

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff d
Merging the files: application

Normal merge conflict for 'application':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

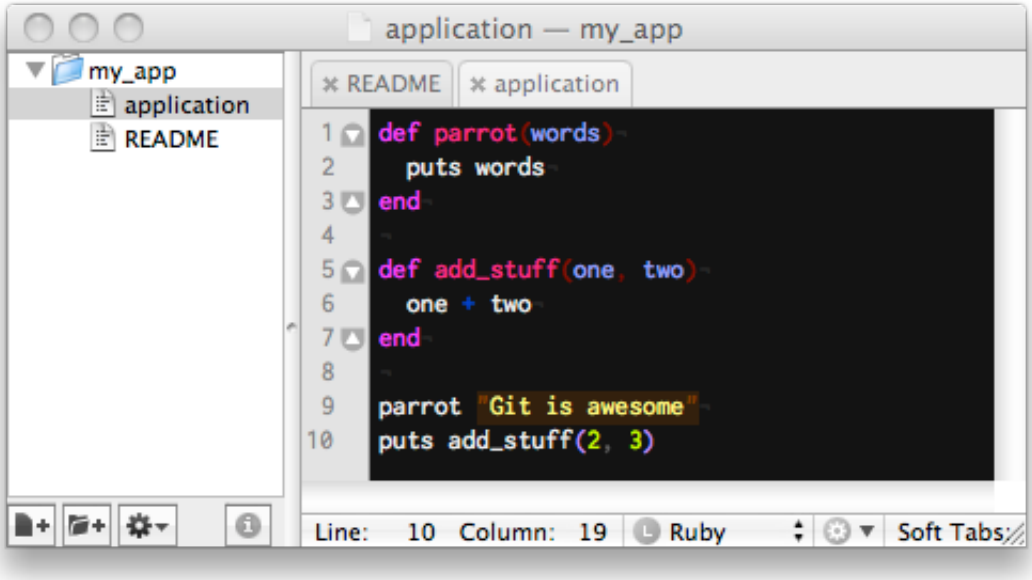
I hit enter and FileMerge pops up and I deal with the conflicts:



All the while git is patiently waiting for the file to be saved. When I save the file (File » Save) and quit the application, git picks up on that. It also leaves a file in the repository called `application.original`. If you're happy with the result of your merge, delete that new file. If not, it's there as a reference while you sort things out (but eventually you need to delete it.) Once you're happy with the merge, commit the changes:

```
1.  $ git commit -a -m 'merged with branch master'
```

Finish our feature: Now that we've merged the fix into our feature branch we can finish what we started so long ago.



And then of course commit:

```
$ git commit -a -m 'finished add_stuff feature'
```

Next we merge it into master so we can deploy it:

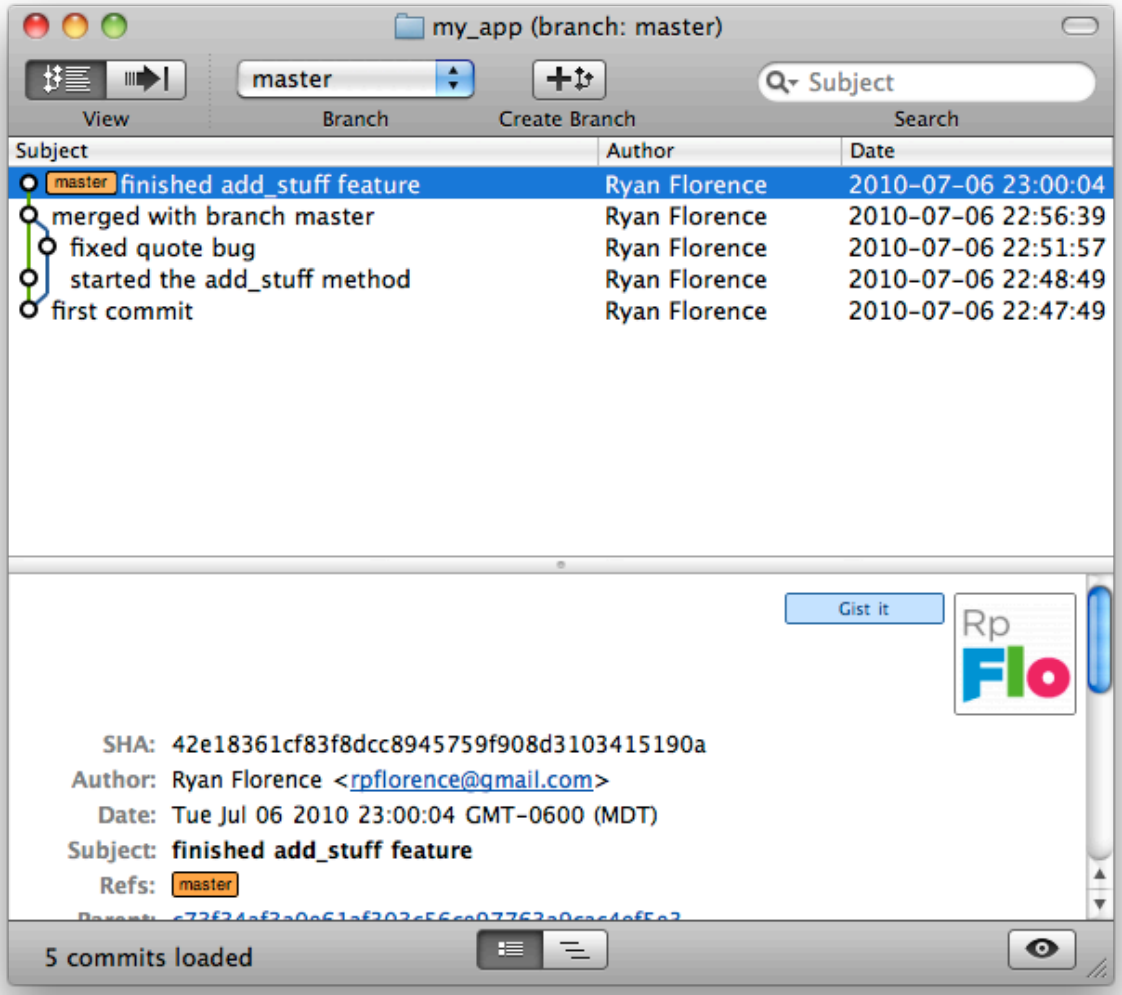
```
$ git checkout master
Switched to branch 'master'

$ git merge develop-awesome-feature
Updating d959c84..3ec8885
Fast-forward
 application |    7 ++++++-
 1 files changed, 6 insertions(+), 1 deletions(-)
```

We can now delete this branch too since we don't need it anymore:

```
$ git branch -d develop-awesome-feature
```

GUI Tools: Git ships with something called gitk, on OS X a lot of folks use GitX instead (I'm in that boat) and on windows, TortoiseGit. Here's a screenshot of what we did throughout this workflow in GitX:

```
$ gitx
```



Git is HUGE. Every week I seem to learn something new that you can do with git. But as you can see, even scratching the surface of its capabilities will make your development far more agile than it would be otherwise. Check out the git resources for other great places to learn.

Richie Zirbes · a year ago

I hit enter and I get this mess:

Deleted merge conflict for 'file.ext':
{local}: deleted
{remote}: created file
Use (c)reated or (d)eleted file, or (a)bort?

Reply · Share ›

gittinOld · 4 years ago

Hi,
Thanx for a great article, was wondering if you could help.

I installed git (git bash) on my windows 7 64 bit computer. Was using happily till I opened the GUI and saw that my whole system was being tracked (system files, the works).

I un-installed Git but am now left with .git folders. Can I just delete those without compromising my system?

I want to re-install Git and use just one dedicated directory next time.

Any help would be most appreciated as I cannot find the right answer anywhere on my google machine :)

Reply · Share ›

Greg Benison · 4 years ago

Great introduction to bugfix branches. In the example, the programmer was fortunate to catch the bug before 'master' had diverged any further; this simplifies merging the bugfix into the feature branch (or into any other appropriate branch). That's why, if master has diverged before discovery of the bug, I recommend creating the bugfix branch from the commit that introduced the bug: http://gcbenison.wordpress.com...

Reply · Share ›

Manuel Neto · 4 years ago

Hi, it,s my first contact on git. I,m a svn user and git is very powerfull.

Good post.

Reply · Share ›

Ciul · 5 years ago

hi Ryan.

I'm finally reading a bit of how to use GIT so will try to upload my plugins to the Forge [MooGeo, MooGooMaps, TwitterRequest]
remember me from MooTools groups? :D

Best regards,
Ciul.

▲ | ▼ · Reply · Share ›

Ciul ➚ Ciul · 5 years ago

Since I'm using windows and I don't like the command line way, I'd prefer a friendly UI...any guide for that?

▲ | ▼ · Reply · Share ›

Ryan Florence Mod ➚ Ciul · 5 years ago

http://code.google.com/p/torto... is what I recommend for command-line averse windows users :)

▲ | ▼ · Reply · Share ›

Ryan Florence Mod · 5 years ago

Thanks for the response 3Easy,

```
rpflo$ git checkout non-existent-branch
error: pathspec 'non-existent-branch' did not match any file(s) known t
```

You can't check out a branch that doesn't exist. That's why you have to toss in the `-b` flag, to tell git to create the branch.

▲ | ▼ · Reply · Share ›

3Easy · 5 years ago

I was pondering the significance of the “-b” in ‘git checkout -b develop-awesome-feature’ and now I know that the “-b” switched you to the new branch as it is created. So ‘git checkout develop-awesome-feature’ is just as valid but then you have to switch to it.

Just saying.
Nice article.
Very nice.

▲ | ▼ · Reply · Share ›

Rolf-nl · 5 years ago

I consider myself a beginner (read: git newbie) and it's always nice to see you are doing/trying "stuff" in a similar way as it is blogged/posted by someone else ;)

▲ | ▼ · Reply · Share ›

Ryan Florence Mod Rolf-nl • 5 years ago

Hi, I'm Ryan!

South Jordan, UT

Location: Github: rpflorence
Twitter: ryanflorence
Freenode: rpflo

About Me

I'm a front-end web developer from Salt Lake City, Utah and have been creating websites since the early 90's. I like making awesome user experiences and leaving behind maintainable code. I'm active in the JavaScript community writing plugins, contributing to popular JavaScript libraries, speaking at conferences & meet-ups, and writing about it on the web. I work as the JavaScript guy at Instructure.