More Git Functionality

Zoe Vernon

10 November, 2020

Section this week is focused on learning more advanced functionally of Git and GitHub. We will practice a number of the tools that are useful when collaborating with others on a GitHub repository. There is additional information about using Git for collaboration and various other features in the appendix at the end for those of you that are curious.

In section we will split into groups of 2 (or 3 if necessary) to practice creating branches and merging.

To Do in Section: Using Git For Collaboration

We will do this as a partner exercise to practice collaborating with git and fixing issues that may arise during collaboration.

1) Create a new repo (just one of you, doesn't matter who)
2) Add your partner(s) as a collaborator
3) Each of you create a new branch, add some files to your branch and practice merging to the `master` branch. You can do either merge directly or use a pull request. See section **Managing Branches*@ below for the commands. Make sure to use `git pull` before merging to avoid merge conflicts.
4) Test what happens when your remote repository is ahead of your local repository, but you already staged new changes.

- Have one partner push a new commit to the remote repo
- Have the other partner try to add, commit, and push new changes and see what happens.
- Resolve the merge conflict. You can do this by calling `git pull` and merging the repositories or by using `git reset --soft` as described in the **Reset** section below.

5) Test what happens when a collaborator (or you on a different computer) edits the same file?

Useful References

Berkeley SCF Git Basics
Software Carpentry Collection of Information on Git
Basic Branching and Merging
Interactive Branching Tutorial
Advanced Merging Undoing Things

Git Tracks Contents, Not Files

Many revision control systems provide an `add` command that tells the system to start tracking changes to a new file. Git's `add` command does something simpler and more powerful: `git add` is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

Manual Pages

You can get documentation for a command such as `git log --graph` with:

```
man git-log
```

or

```
git help log
```

Viewing Project History

At any point you can view the history of your changes using:

```
git log
```

If you also want to see complete diffs at each step, use

```
git log -p
```

Often the overview of the change is useful to get a feel of each step:

```
git log --stat --summary
```

For a prettier, more detailed graph (with several more options to look up):

```
git log --oneline --decorate --graph --all
```

Undoing a Mistake: checkout, reset, and revert

Checkout

`git checkout` can be used to look at a previous commit. It can also be used to move to a different branch, which we will look at in the next section. Here we can look at code from a previous commit with:

```
git checkout HEAD~1 # moves back 1 commit
git checkout HEAD~2 # moves back 2 commits
git checkout <commit_hash> # move back to a specific commit
```

To find commit IDs you can use `git log` or `git reflog`. You can also find commit IDs on GitHub.

Once you have looked at the commit you can go back to the most recent update using

```
git checkout master # or replacing master with whatever branch you are on
```

Revert

`git revert` is used when you want to undo the changes made in a previous commit. It will undo a commit by creating a new commit. Consider using `git revert HEAD~1`, this will remove the changes that were added in the previous commit.

```
git revert HEAD~1
git revert HEAD~2
git revert <commit_hash>
```

Reset

If you added something that shouldn't be commited or you want to reset your repo to what it looked like at a previous commit, then you need to use the `git reset` feature.

```
man git-reset

# e.g.
git reset --soft HEAD~1
git reset --hard HEAD~1
```

Git reset moves the tip of your working tree back to the specified revision (here, we go back one revision). The `--soft` flag means that the changes in the files are presevered, so all that was done was to undo the commit. If you use the `--hard` flag, then all changes are reverted to the specified time.

If you have done something more complex, like commiting and attempting to push a large file, the command below may be useful.

```
git filter-branch --index-filter 'git rm -r --cached --ignore-unmatch <PATH/TO/FILE>' HEAD
```

This deletes everything in the commit history.

Managing Branches

A single Git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
git branch experimental
```

If you now run

```
git branch
```

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
git checkout experimental
```

to switch to the experimental branch. Now edit a file, commit the change, and switch back to the master branch:

```
git add file
git commit -m "editted file"
```

Alternatively, one can "stash" the changes using `git stash`. This saves your changes for later, and then reverts the working tree to the last HEAD (whatever the last commit was). This allows you to keep working without the changes being applied to any files. You can apply those changes later using `git stash pop`, which applies the changes and removes them from your stash. Or, if you wish to apply the changes to multiple branches, you can use `git stash apply`, which applies the changes but leaves them in your stash.

Now, you can switch back to the master branch.

```
git checkout master
```

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch.

You can make a different change on the master branch and commit. At this point the two branches have diverged, with different changes made in each. To merge the changes made in experimental into master, run:

```
git merge experimental
```

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
git commit -a
```

will commit the result of the merge. Finally,

At this point you could delete the experimental branch with

```
git branch -d experimental
```

This command ensures that the changes in the experimental branch are already in the current branch.

If you want to remove a branch without pulling the changes into the master branch, the `-D` flag deletes it without checking any of the changes.

This only removes the branch from your local machine. To remove it from the remote repository, you use:

```
git push -d origin experimental
```

Pull-requests

When working in a collaborative environment, instead of merging directly into the master, it is better to create a pull-request. This link has a good step-by-step explanation. Pull requests tell repository maintainers the difference between the master repository and an individual's branch. It will then allow maintainers to comment on the pull request and get bugs fixed before the branch is merged to the master.

Pull requests are common practice in software development in industry.

Appendix: More Useful Git Functionality

Importing A Project

I **do not** recommend this process for initiating a new project. These steps are simple, until you get to creating the remote repository. Then, just like in the intro tutorial, you have you setup a new repository on Github and link it to the local one. It is easier to create the repo on Github, clone the empty repo locally, then put files in it as desired.

Assume you have a tarball linReg.tar.gz with your initial work. You can place it under Git revision control as follows.

```
tar xzf project.tar.gz
cd project
git init
```

Git will reply (something along the lines)

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

You've now initialized the working directory-you may notice a new directory created, named ".git".

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with git add:

```
git add .
```

This snapshot is now stored in a temporary staging area which Git calls the *index*. You can permanently store the contents of the index in the repository with `git commit`:

```
git commit -m "add"
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

Making Changes

If we make changes to files `file1`, `file2` and `file3` we can add them to be commited with `git add` as we have discussed before:

```
git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using `git diff` with the `--cached` option:

```
git diff --cached
```

(Without –cached, git diff will show you any changes that you've made but not yet added to the index.)

You can also get a brief summary of the situation with `git status`:

```
git status
```

Alternatively, instead of running `git add` before `git commit`, you can use:

```
git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

Amending file to commit

What if, in the files you just commited, there was a file you forgot? This situation is handled via git's `amend` option in git commit. Say we have added and committed `file1`

```
git add file1
git commit -m "adding file1"
```

But we realize we also meant to commit `file2`. We can do that by ammending the original commit as follows:

```
git add file2
git commit --amend -m "adding second file"
```

This allows you to add more files to a commit and then update the message, while keeping your original message/commmited-files there.