Quick Reference for Git Processes on GitHub
Chris Ahlstrom
2015-09-09 to 2024-04-17

Creating a GitHub Repository

First, create the project on the computer. The steps are slightly different, depending on whether there is yet no code versus a body of code outside of the control of *git*. The first set of commands refer to an empty project. Once project directory is in place, run the following commands in the top directory of the project. If the project is unpopulated:

```
$ git init
$ git add .
$ git commit -m "Created this new and wonderful project."
```

The first thing to do is to make sure to create the following files: .gitignore, README.md, LICENSE.md, and NEWS. If the project already has a number of files ready, then do the following instead:

```
$ git init
$ git add *
$ git status        # verify that all project files are shown in this list!
$ git commit -m "Created this project using the existing files."
```

In *GitHub*, create the new repository (without the LICENSE and README files, which one can create locally and in a better format, markdown).

Then copy the repository URL to the system clipboard. We will call it $*MY_GITHUB_URL*. It is important that the SSH URL (e.g. git@github.com:ahlstromcj/seq66.git) be used, since username:password access no longer works in *GitHub*. Add the remote URL, verify it, and push this local repository to it.

```
$ git remote add origin $MY_GITHUB_URL
$ git push -u origin master
```

In *GitHub*, verify that the default branch is master and that it contains any files that were added in the steps above. Now we are ready to work, making changes locally and pushing them to GitHub.

Create a Repository on a Local Server

On a side note, if one wants to create a local repository called "proj.git" on a machine with many users, who are all in the group "devs", make and change to a directory such as /srv/git, then initialize the repository to be "shared":

```
# mkdir /srv/git
# cd /srv/git
# mkdir proj.git
# git init --bare –shared=devs  proj.git
```

The *--bare* option avoids creating a working directory, which is desirable if the repository is to be shared. The *--shared* option makes the repository group-writable and *g+sx*. In the repository *.git/config* file, make sure to find:

```
[core]
bare = true
sharedrepository = 1                # i.e. "true"
```

Then change the group owner of the repository, if not already set to "devs" by git:

```
$ chgrp -R devs proj.git
```

To get the code to this new bare repository, first go to the top-level directory in the code base (e.g. the "proj") directory. Then:

```
$ git init
$ git add .    (or “git add *”)
$ git commit -m "Initial commit before pushing to central repo."
$ git remote add origin ssh://user@theserver/srv/git/proj.git
$ git push --all origin   -or-   git push -u origin master
```

This process is very similar to what done does for *GitHub*. A simpler process can be used if one just wants to share a home repository between various local *Linux* computers where SSH has already been set up with keys.

Main computer (server):

```
$ git init
$ git add .
$ git commit -m "Created this new and wonderful project."
```

Other computers:

```
$ git clone ssh://user@maincomputer/srv/git/proj.git
```

Repository Branch Management

https://github.com/Kunena/Kunena-Forum/wiki/Create-a-new-branch-with-git-and-manage-branches and others.

In a GitHub fork, keep the master branch clean (no changes pending). Then one can create a branch from the master. Each time one wants to commit a bug or a feature, create a branch for it, which is a copy of the master branch.

Create the branch on the local machine and switch to it:

```
$ git checkout -b [name_of_your_branch]
```

To branch from another branch, add it to the command:

```
$ git checkout -b [name_of_your_branch] [name of existing branch]
```

Here is an example of a feature branch from the tutorial https://nvie.com/posts/a-successful-git-branching-model/:

```
$ git checkout -b myfeature develop # Switched to a new branch "myfeature"

# Finished features may be merged into the develop branch to definitely
# add them to the upcoming release:

$ git checkout develop             # Switched to branch 'develop'
$ git merge --no-ff myfeature      # Updating ea1b82a..05e9557
$ git branch -d myfeature          # Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

When the branch is ready, push the branch to the "origin" remote on GitHub as in the example above:

```
$ git push origin [name_of_your_branch]
```

Better, to make sure git pull will work, set an upstream branch:

```
$ git push -u origin [name_of_your_branch]
```

If *git* complains about the branch having no upstream branch, then:

```
$ git push --set-upstream origin [name_of_your_branch]
```

When doing a pull request on a branch, one can continue to work on another branch and make another pull request on this other branch. To determine what changes were made in a pull request, use the following command. To just list the files, leave off the -p.

```
$ git request-pull -p master https://github.com/ahlstromcj/seq66.git
```

When you want to modify and commit something in the branch, first checkout the branch. If *git* claims it cannot check out a branch, do the following command to tell the local *git* repository about new branches:

```
$ git fetch
```

Update the branch when the original branch from official repository has been updated:

```
$ git fetch [name_of_your_remote]
```

(It is important to be aware that the fetch command does not do a merge, while a pull command does attempt to do a merge. Doing a pull of one branch while the current branch is a different one is something one generally does not want to do.)

Others can track the branch in this manner to simplify the git push command:

```
$ git checkout --track -b your_branch origin/your_branch
$ git commit -m "the relevant message"
$ git push
```

Then apply (merge) the changes. If the branch is derived from the "develop" branch do this action:

```
$ git checkout master
$ git merge [name_of_your_remote]/develop
```

If one had set up an upstream branch, then this command will suffice:

```
$ git merge develop
```

One can add the --no-commit option to allow for inspecting the merge and doing further tweaking of the merge before committing the merge.

Then make any minor tweaks or version stamping needed to make the master official. Also, it is beneficial to tag the release (as discussed below) to make it easier for users to determine the differences between major/minor releases, without having to study a bunch of intermediate commits.

One can also merge from master to a feature-branch without a checkout:

```
$ git merge master feature-branch
```

One will probably also want to merge back any tweaks made to master back into your branch, if you intend to do further work in that branch:

```
$ git checkout develop
$ get merge master
$ git push origin develop
```

Branch delete/remove

First delete the branch on github:

```
$ git push origin :name_of_branch           -or-
$ git push origin --delete name_of_branch
```

Follow that up with this command on all the local copies:

```
$ git fetch --all --prune                  -or-
$ git branch -d name_of_branch
```

The prune command has failed on one of our systems with a Python error regarding *apt_pkg*. Even if *python-apt* or *python3-apt* are installed.

Caching ones credentials (be careful!)

```
$ git config credential.helper store
```

Pulls and merges

For a GitHub project, one can use the handy "Merge pull request" button on the project page, or do it from the command line.

Step 1: From the project repository, check out a new branch and test the changes.

```
$ git checkout -b TDeagan-mute_groups master    # or a branch
$ git pull https://github.com/TDeagan/sequencer64.git mute_groups
```

Step 2: Verify that the changes are reasonable and correct, building if needed. The following command will show all the changes, and one can build and test if the changes look suspicious.

```
$ git diff --name-only master                   # How many files changed?
$ git difftool master                           # View them (e.g.  gvimdiff)
```

Step 3: Merge the changes and update on GitHub.

```
$ git checkout master
$ git merge --no-ff TDeagan-mute_groups
$ git push origin master
```

Trial merge

To do a trial merge, pass in the --no-commit flag, but, to avoid a fast-forward commit, also pass in --no-ff, like so:

```
$ git merge --no-commit --no-ff $BRANCH
```

To examine the staged changes:

```
$ git diff --cached
```

And one can undo the merge, even if it is a fast-forward merge:

```
$ git merge --abort
```

This still modifies the working copy.

Rules for a Git commit message

- Do NOT end the Subject line with a period.
- Separate Subject from Body with a BLANK LINE.
- Capitalize the Subject line and each paragraph.
- Wrap lines at 72 characters.

Making a GitHub Release

Steps for making a release:

1. Make sure the working branch builds and runs properly for the *rtmidi*, *portmidi*, debug, Windows, and command-line versions, and all changes are pushed to GitHub.
2. Run "./bootstreap --full-clean".
3. Check out the "master" branch and merge the working branch into it.
4. Make sure the version numbering and dates are current in *configure.ac*, *VERSION*, *README.md*, *RELNOTES*, i*nclude/qt/portmidi/seq66-config.h*, *include/qt/rtmidi/seq66-config.h*, *seq66-user-manual.tex*, *data/readme.**, *data/license.text*, and the NSIS and batch files.
5. Update the *README.md* and *RELNOTES* files and the *seq66-user-manual.tex* files for version and dates, and any other updates that reflect important changes for the new release. See below.
6. Verify the master branch builds ("*./bootstrap -er*" followed by "*make*") and runs.
7. Update the files in the "*nsis*" directory to the latest version, and update the *readmes* and *license* files as necessary.
8. Remove any temporary/test tunes from *data/midi*. (Test files in *contrib/midi* are okay, they are not packaged in a release.)
9. Do the command *./pack windows* to create a *7z* file that can be extracted in the Windows build machine.
10. Verify that the Windows version builds, runs, and that the Windows installer is created and works, by uninstalling the current version and running the installer for the new release version. Verify the configuration files in *C:/Users/user/AppData/Local/seq66*. If desired, delete them all and recreate, then verify them. Update them for any MIDI port issues. Copy files from *data/samples* if desired to test palettes, style-sheets, and MIDI contro/display functionality.
11. Copy *seq66-release-64/Seq66qt5/qpseq66-release-package-x64-0.99.6.7z* (the portable "Zip" file) and *seq66/release/seq66_setup_x64-0.99.6.exe* (the 32-bit NSIS installer for the 64-bit *qpseq66.exe*) to the top directory or some other safe place for later upload to GitHub.
12. Go to "*doc/latex*" and run "*make*" to rebuild the PDF manual.
13. Run "*git2cl > cl.txt*" and use it to update the *Changelog* file.
14. Commit this release build *without* the "-m" option. Instead, read the *RELNOTES* file into the commit message. See the commit-message rules above. Also update the *NEWS* file appropriately.
15. Push the release.
16. Get the HEAD checksum and use it to create the tag such as "0.99.6" with a simple message about the version number. Push this tag to GitHub.
17. In GitHub, update the "About" message with the version and date.
18. On GitHub, create a release with a name of the form "*Fancy Name 0.99.6*". Do not forget the version number!
    1. Type a short but complete description of the main features of the release. A good option is to create a *RELNOTES* file, add the release description to it, and copy the text to it. (And update the *NEWS* file at the same time.)
    2. Add the Windows installer *seq66_setup_x64_0.99.6.exe* and *qpseq-release-package-x64-0.99.g.7z* to the release packages.
    3. In the "*Choose a tag*" dropdown, select the created tag (e.g. 0.99.6).
    4. Click the "*Set as latest release*" button.
    5. Click the "*Publish release*" button; verify that it shows up in the main "seq66" page.
19. Follow up with any comments or closures of issues.
20. Go to https://github.com/ahlstromcj/ahlstromcj.github.io/tree/main/docs/seq66 and upload the latest version of the PDF user manual.

Tagging

We want to tag release points (e.g. 0.99.9) with a tag of the same name. We create an annotated tag, which adds information and requires a commit message, and is meant for public consumption.

```
$ git log              # to get the commit hash, abcd1234 (for example)
$ git tag -a 0.99.9 -m "Version 0.99.9" abcd1234
$ git push origin 0.99.9                - or -
$ git push origin --tags
```

If one tags the wrong commit, simply re-tag with the correct commit, and add the "-f" (force) option.

The GitHub "Create Release" feature can be used. Install the "hub" app; it is the official command-line version of GitHub. (It also pays to install "gitsome", which adds the "gh" application also useful in creating releases. Then:

```
$ hub release create --copy -F RELNOTES v2.3.0
$ hub release create -a 0.9.9  -F RELNOTES v2.3.0
$ gh release -F RELNOTES create <tag> <assetfiles>
```

Here is what we will use:

```
$ git config --global hub.protocol ssh
$ git tag -a 0.99.0 -m "Version 0.99.0" abcd1234
$ git push origin --tags
$ hub release create 0.99.0 -F RELNOTES abcd1234
```

For the first command, the URL of the new release is copied to the clipboard. See https://hub.github.com/ for more information. For example, one can start a discussion with gh.

Each tag to be published needs to be pushed as well. Also note that, if the commit hash is missing, HEAD is assumed.

```
$ hub release create -a prebuilt.zip -m 'release title' tag-name
```

- Prompts for your password the first time, and then automatically.
- Creates and stores an API token locally.
- Creates a non-annotated tag on the remote called tag-name.
- Creates a release associated to that tag.
- Uploads prebuilt.zip as an attachment.

Tag Checkout

A user can get the tagged code into a new branch via commands like the following. First, fetch tags from the remote repository. Then check out the desired tag (here, "0.00.0" into a new branch with the desired name:

```
$ git fetch --all --tags
$ git checkout tags/0.99.9 -b newbranch
```

If a new branch is NOT desired (i.e. one wants to compare an old version with a current buggy version in a separate copy of the repository), use this:

```
$ git checkout 0.99.9
```

Be careful about modifications; if committed, they represent new work one might need to merge later, but the new work will not be part of the tagged commit.

To get all of the tags for a project, a simple pull (of all items) is necessary:

```
$ git pull
```

To show all of the tags present in the project, along with the commit numbers for the tag and the tag operation:

```
$ git show-ref --tags
```

To show just the tag names:

```
$ git tag
```

To add the date of each tag:

```
$ git tag -l --format="%(creatordate:short)|%(refname:short)" | sort -r
```

Related to tags, one can generate a brief description of the current commit, based from the latest tag, using the following call (with the result shown):

```
$ git describe --tags --long
0.9.11-12-gca37826
```

This output says that the command was run after the 12th commit after the 0.9.11 tag, and includes a hash code (after the "g", which simply denotes that the "git" DVCS is being used) that can be used if the commit changes, or there are many 12th commits. Note that the output also depends on the current branch that is checked out. Also, there is an --always option that can be added to grab the latest hash value found even if no commit has yet been tagged (as is the case early on in a project).

To check out a specific tag:

```
$ git fetch --all --tags --prune
$ git checkout tags/<tagname> [-b branch name]
```

The branch name is optional; one might not want to make a branch, and just get the tagged version.

```
$ ./pack 0.94.6
```

This will pack up the code and Makefiles as a tar-file that can be used to ./configure and build the code.

To remove a tag:

```
$ git tag -d 0.91.5
$ git push --delete origin 0.91.5
```

Cherry-Picking

If one makes a fix in a branch and wants just that commit to be merged (into master), then do the following after committing the fix in the branch, assuming the branch and master are both now clean:

```
$ git log                   # and record the last commit hash
$ git checkout master
$ git cherry-pick --edit a2d34fb
```

And then one can push the fix and the new tag.

See this article on why one should generally avoid cherry-picking in git:

http://www.draconianoverlord.com/2013/09/07/no-cherry-picking.html

Pull Requests

When one gets an email about a pull request, there are a couple of ways of dealing with it, given that it is an acceptable request.

First, one can go to the project GitHub page, and click on the "Merge pull request" button. Second, one can do something like this, from the command line:

```
$ git checkout -b otherproject-master master
$ git pull https://github.com/otherproject/seq66.git master
$ git checkout master
$ git merge --no-ff otherproject-master
$ git  push origin master
```

Diffs

Summarizing differences between commits, much like "svn diff --summarize":

```
$ git diff --name-status <commit1> <commit2>
```

Then, if one has the following in the .gitconfig file:

```
[diff]
    tool = gvimdiff
[difftool]
    prompt = false
```

One can cycle through a series of diffs on the involved files with one of the best diff viewers ever:

```
$ git difftool <commit1> <commit2>
```

Getting a list of files that have conflicts:

```
$ git diff --name-only --diff-filter=U
```

Log information

We add to our *ChangeLog* using the Perl script "git2cl":

```
$ git2cl > cl.txt
$ gvim -O cl.txt ChangeLog
  (copy stuff from cl.txt to ChangeLog and clean it up)
```

A quick way to summarize the files edited in the last few commits is:

```
$ git log --pretty --numstat --summary
```

Listing files

One often wants to do a quick edit or survey of modified files. The following command is useful:

```
$ git ls-files -m
```

In bash, one can pass the list to vi(m):

```
$ vi $(git ls-files -m)
```

To list only untracked files:

```
$ git ls-files --others --exclude-standard
```

From *https://jayenashar.wordpress.com/2014/03/07/git-status-sorted-by-last-modified-timestamp/*, here's a command that will show the modified, but unstaged files, sorted by time:

```
$ ls -lrtd $(git status --porcelain | grep '^.[?M]' | sed 's/^.. //')
```

Listing issues

See *https://adriansieber.com/download-github-issues-from-command-line/* Browsing to the following URL will obtain a dump of all Seq66 issues: *https://api.github.com/repos/ahlstromcj/seq66/issues*). This dump needs programming to whip it into human-readable form.

```
$ http -b https://api.github.com/repos/ahlstromcj/seq66/issues?state=all
```

Permanently delete a file and its history

```
$ git filter-branch --index-filter \
  "git rm -rf --cached --ignore-unmatch seq66/0.90/seq66_setup_0.90.4.exe" \
  HEAD
$ git push --all
```

This can take a long long time if the file has thousands of commits. Not yet sure if this really works!

Git Submodules

Submodules allow one to keep a Git repository as a subdirectory of another Git repository; one can clone another repository into a project and keep the commits separate. One can add an absolute or relative URL of the project to track:

```
$ cd my_git_repo
$ git checkout master
$ git submodule add https://github.com/chaconinc/DbConnector
```

This creates a .gitmodules file (which is tracked in Git) and a DbConnector directory.

```
$ git commit -am "Add SbConnector project as a submodule."
$ git push origin master
```

To get an existing project that has submodules:

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject
```

For details, see https://git-scm.com/book/en/v2/Git-Tools-Submodules, it discusses many more submodule commands.

To update:

```
$ git submodule update --remote DbConnector
```

Leave off the submodule name to update all of them. To pull upstream changes:

```
$ git pull
$ git submodule update --init --recursive
```

The "--init" covers the case that the main project added more submodules. To push submodules, and then the main project, use:

```
$ git push --recurse-submodules=check
```

To run a command for all submodules:

```
$ git submodule foreach 'git stash'
$ git submodule foreach 'git checkout -b NewFeatureA'
```

To delete a submodule:

```
$ git submodule deinit -f -- DbConnector
$ rm -rf .git/modules/DbConnector
$ git rm -f DbConnector
```

Creating a Personal GitHub Site

1. In the upper-right corner of any page, use the "+" drop-down menu, and select "New repository". [Note: actually, go to the user-profile link (your face), then the "Your repositories" link. Then click the "New" button.]
2. Enter *username.github.io* as the repository name. Replace username with the GitHub username.
3. Under your repository name, click "Settings".
4. In the "Code and automation" section of the sidebar, click "Pages".
5. Click "Choose a theme". Browse and pick a theme.
6. Modify the new "*README.md*" file. Click "Commit changes."
7. The site can be changed by editing "_config.yml" in the "Code" tab. Click "Commit changes".

Note that is can take up to 20 minutes for site changes to appear. See *https://docs.github.com/en/pages/quickstart* for "Next Steps".

Adding a PDF to the personal GitHub Site:

Once the site is created, clone it locally. The PDF needs to go into https://username.github.io/folder.

Cloning our personal GitHub site:

```
$ git clone https://github.com/ahlstromcj/ahlstromcj.github.io
```

Then it is good to use the existing SSH/Git access by editing *.git/config* to set:

```
[remote "origin"]
    url = git@github.com:ahlstromcj/ahlstromcj.github.io
```

Otherwise, GitHub now gripes about security and refuses to deal with a *https* link.