Using GitHub with RAT-PAC Code

Andy Mastbaum[*]

December 17, 2014

Contents

1 Introduction

Using Git and GitHub makes it easier to share code and allows review to happen before code is committed to the main repository, ensuring that a high-quality code base is always available to collaborators. This document introduces Git and GitHub and provides a walk-through of typical developer workflow. A "cheat sheet" suitable for hanging near a computer is attached at the end.

[*]mastbaum@hep.upenn.edu

1.1 What is Git?

Git is a distributed version control system, originally developed by Linus Torvalds and used for Linux kernel development. Now, it is used by such prominent projects as Perl, Gnome, Android, X11, and RAT-PAC.

Distributed version control systems (DVCS) are next-generation version control systems (VCS) structured with a network of peer repositories, rather than a single "master" repository. In SVN (a regular VCS) for example, users "check out" a snapshot of a repository, make changes, and push the new snapshot to the master repository every time they wish to change something. In Git (DVCS), users "clone" a repository, creating a complete replica of it on the user's computer. The user can commit to this repository at will, view the commit history and "check out" any past revision offline, and later synchronize their changes with the parent repository.

This model allows great flexibility in the flow of code changes. It makes it easier for developers to collaborate on features, and also makes it possible to create a "staging area" for code review.

1.2 What is GitHub?

GitHub is a Git repository hosting service that offers a suite of best-in-class source code management tools. GitHub encourages cloning (called *forking*) of projects for collaborative development. In the past, fork meant divergence: a group of developers broke off from the project to pursue different direction. Now, thanks to Git's distributed nature, forking means collaboration: you see a project that needs help, fork it, make changes, and give those changes back to the original repository. GitHub makes this process as easy as a few clicks.

The most important feature for RAT-PAC is the *Pull Request* system, the mechanism by which developers "give those changes back" from a fork to the parent repository. Pull Requests provide a chance for code to be reviewed, discussed, and improved before it is committed to the main repository.

GitHub also offers web-based code browsing, search, and project management tools including a bug tracking system that integrates with release management.

2 User's Guide

This section explains how to get started with RAT development. If you just need to download RAT on a computer, see Section 3.1 for a shortcut.

2.1 Getting Started

RAT-PAC is publically-accessible, open-source software, so you don't need a GitHub account for read-only access. Write access is available to members of collaborations affiliated with RAT-PAC development. If you are such a collaborator, you may request access as follows:

1. Make yourself a free account here: https://github.com/signup/free
2. Send a message to Gabriel and Andy asking to be added to the rat-pac organization. Include your full name, GitHub username, insititution, and experimental affiliation, and specify that you agree to the policies as laid out in the shared development plan document.
3. Once you've been added, you will have write access to the repositories at github.com/rat-pac.

Now you can browse the code on GitHub, authenticating using your username and password. But to access the code via the command line, e.g. to download it, your Git client will need to authenticate too. This is best done with SSH keys.

If you are familiar with SSH keys and already have one, just to https://github.com/settings/ssh, click "Add SSH Key", and paste in your public key. Otherwise, GitHub has a quick tutorial on how to set this up: https://help.github.com/articles/generating-ssh-keys.

2.2 Creating a Fork

Now, you need to create a *fork* of the main RAT repository. Your fork is a duplicate of the main repository which you can work on in isolation, even doing your own project management with bug tickets and milestones. When you're ready, you can send a Pull Request, asking to have your changes merged into the main repository. Typically you will make many commits, incrementally working toward a new feature, between Pull Requests. This helps you keep track of every step (and rewind if things go wrong!)

Your fork is a proper repository unto itself. Other people can fork it, make changes, and send *you* Pull Requests, which you can review and accept, then pass them along as part of a Pull Request to the main RAT repository. The collaborative possibilities are endless, and in every case, Git keeps track of all the history of who changed what, when, and why.

To fork a repository (be it RAT or any other), navigate there on GitHub, and click the "Fork" button in the upper right corner. Your fork will now appear in your list of repositories on your GitHub page (github.com/[your-username]). You should only ever do this once per repository: work is done in many *branches* within a single forked repository, rather than in many forks.

After you click the fork button, your fork only exists on GitHub's servers, and is identical to the main RAT-PAC repository until the RAT-PAC repository changes – this is never propagated automatically!

2.3 Cloning Your Fork

The command to clone a repository is[1]:

```
$ git clone URL [optional destination]
```

In this case, you'll want[2]:

```
$ git clone git@github.com:[your-username]/rat.git
```

This puts RAT in a new directory called `rat`. To put it somewhere else, tack the destination directory name onto the end of that line. If there is a particular branch you are interested in, do a

```
$ git fetch
$ git checkout BRANCH_NAME
```

to switch to it.

To install and run RAT after running `git clone`, you'll need to set up ROOT and GEANT4 as well, source their respective environment scripts (`$ROOTSYS/bin/thisroot.sh` and `$G4INSTALL/bin/geant4.sh`), then run (in the rat directory):

```
$ ./configure
$ source env.sh
$ scons
```

to build RAT.

Don't Forget Your Roots You'll want to pull in the changes from the main RAT repository periodically, to stay up-to-date and avoid conflicts when the time comes to contribute your changes. To do this, we need to set up a pointer to main repository – a remote repository or *remote*. The basic syntax is:

```
$ git remote add NAME URL
```

where `NAME` can be anything you like, and `URL` is a Git URL like the one we cloned. Here, use:

[1]For complete usage, try `git help clone`. `git help X` always brings up the man page for subcommand `X`.
[2]The URL you need to `git clone` to clone a repository always appears at the top of that repository's page on GitHub.

```
$ git remote add upstream git@github.com:rat-pac/rat-pac.git
```

This makes a pointer to the main RAT repository and calls it 'upstream' (this is conventional, but has no special meaning to Git).

You can also add other peoples' forks as remotes in the same way to pull in their changes. The command `git remote -v` prints a listing of all the remotes.

The remote that you used in the initial `git clone` command has a name, too: `origin`.

2.4 Code Management

Your fork is yours to manage as you like, but following a few guidelines will make things much smoother when interacting with the main RAT repository.

2.4.1 Never Commit Changes to Your 'master' Branch

The default "main" branch in a Git repository is called "master," and it is strongly suggested that you keep your master branch in sync with the upstream master branch. Instead of modifying master, create *topic branches* and do your work there. Creating branches in Git is simple; to create a new branch and make it active:

```
$ git checkout -b BRANCH-NAME
```

Try to keep topic branches independent – avoid developing one branch that depends on another if possible. It will cause headaches if the base branch is not accepted into main RAT or requires substantial changes.

2.4.2 Merge Upstream Changes Often

Development in the main repository may happen fast, and if you work in isolation for a long time, you will find that the changes you've made are no longer compatible with the main version of RAT and cannot be merged in. To prevent this, stay up to date and resolve small conflicts as they occur. If you set up the 'upstream' remote as outlined above, run:

```
$ git fetch upstream         # grab the changes from the remote
$ git merge upstream/master  # merge changes into the current branch
```

Handling Conflicts It is possible that changes have been made to the main repository that conflict with your local changes. Git makes an effort to resolve conflicts automatically, but in some cases (like edits to the same line), it can't tell what is correct. In this case, `git merge` will keep both versions, so that the files look like this:

```
<<<<<<<
Version in the current branch (yours)
=======
Version in the branch being merged in (theirs)
>>>>>>>
```

It will tell you which files have conflicts. Go and fix them by hand, choosing the correct version or editing as necessary, then mark the conflict as resolved by running `git add FILENAME`. Once all conflicts are resolved, you need to complete the merge by committing:

```
$ git commit -am "merge branch upstream/master"
```

Remember: just because Git conflicts are resolved does not mean that the code will actually compile and run! Be sure to check before committing your merge.

There are other ways of handling merge conflicts – see `git help merge` for information.

2.5 Viewing Status and Changes

Git has a few commands to keep track of changes:

`git status` Shows which files have been modified, added, and removed

`git diff` Shows a summary of all changes since the last commit

`git diff X` Shows outstanding changes relative to *ref* X

`git diff X Y` Shows differences between *ref* X and Y

A *ref* in Git is a pointer to a commit. This can be a revision ID (the long hexadecimal identifier), a tag name (like `1.0` for an imaginary RAT 1.0), a branch name (like `master`), or one of a handful of special identifiers (like `HEAD~5` for five commits ago).

2.6 Committing Changes

A *commit* packages all of the outstanding changes into a *changeset* which becomes part of the history of the repository. Each commit is accompanied with a *message* – a description of what the changes are and why they were made.

Git only records changes for files when it's told to, so if you have created new files you need to do:

```
$ git add FILENAME
```

to inform Git that this file should be tracked.

Now, to commit the changes (including newly-added files and modifications to existing ones):

```
$ git commit -a
```

Remember not to commit to 'master'! The `-a` here means to commit **all** the outstanding changes shown by `git status`. To instead commit individual files, use `git commit file1 [file2 ...]`.

This command will pop open your system's default editor so you can type in a commit message (see Section 2.8 for best practices). To change the editor, set the `EDITOR` environment variable, e.g. `$ export EDITOR=vim`. For a short message, you can also do `$ git commit -am "message here"`.

Now, the changes are stored in the repository on your computer. Periodically, you should *push* your changes to the GitHub server, which will serve as a backup and also let people see what you're working on. Eventually, all changes must be pushed to GitHub in order to be merged into the main repository, but you don't need to push after every commit. To push changes, the command is:

```
$ git push [REMOTE NAME] [BRANCH NAME]
```

So if you are working on a topic branch called 'awesome-feature', use:

```
$ git push origin awesome-feature
```

This creates and populates the branch 'awesome-feature' on GitHub's servers if it doesn't already exist. If you run `git push` with no arguments, Git will push all the branches that match existing remote branches[3]. It's probably best to just specify explicitly the branch you want to push.

Remember that `origin` is the name of the remote used in `git clone`, i.e. your fork.

[3] You can configure the default behavior with `git config`.

2.7 Documenting Changes

RAT features should be documented in the RAT User's Guide. When you submit a pull request to the RAT repository that adds a new feature or changes the way RAT works, you should always be sure to update or add to the documentation in the same pull request. Pull requests that neglect the documentation are likely to be rejected!

This documentation, written in Sphinx-style ReStructured Text (ReST), lives in the `doc` subdirectory of the rat-pac repository. For help with writing in ReST, see http://sphinx-doc.org.

The User's Guide is compiled into web-accessible HTML after each commit to the main RAT-PAC repository; this can be found at http://rat.rtfd.org.

2.8 Best Practices

There are also a few tips to follow to maximize your developer karma:

2.8.1 Write Good Commit Messages

The commit message should record what was changed and why. The proper format for a Git commit message is:

```
Describe changes briefly (50 chars or less)

After a blank line, provide a detailed, multi-line description,
wrapped to 72 characters.

 - Bullet points are nice
 - ... if you have things to list
```

GitHub will format these messages nicely, and Git fanatics are *rabid* about commit message formatting. Also note the use of present-tense verbs: *edit* the file, *fix* the bug.

2.8.2 Keep the History Tidy

On one hand, the history should be a complete record, but on the other hand, extraneous changes (e.g. changed thing, reverted change, reverted revert, ...) clutter the history and make it very difficult to follow.

As a guideline, a typical pull request should have one or a few commits. If you prefer finer granularity (lots of tiny commits) when developing, use Git's "squash" merge to combine your changes into a single changeset with a meaningful message. Say you've developed a feature on branch 'mybranch':

```
  $ git fetch upstream
  $ git checkout -b mybranch-squash upstream/master  # make new branch from the latest master
  $ git merge --squash mybranch  # merge in your changes all at once
  $ git commit  # commit the merge. the editor will give you the list of individual messages
```

2.8.3 Don't Rebase

Git offers the powerful but deadly `git rebase` command which can modify the Git history. This will in general render the rebased branch incompatible with other branches, and should be avoided unless you are certain that the affected commits aren't in use by anyone else.

2.9 Contributing Your Changes

At last, the feature is done, the bug is fixed, the document is documented; you're ready to push your changes off into the main repository. First, consider:

- Is the change appropriately documented?

- Does the code comply with the style conventions of RAT?
- Does the code introduce any new external dependencies?
- Have any relevant `rattestss` been updated?

Push all of your changes to a branch on your GitHub fork. Then to send it off for review, click the "Pull Request" button at the upper right corner of your fork's repository page on GitHub. On the Pull Request page, you can choose the correct topic branch and review the changes. Enter a title and description of your changes and click "Send pull request."

This will notify everyone following the the main repository, and someone will take a look over the changes. If you're contributing an experiment-specific feature, ask a qualified person to do the review and merge the changes.

If anything's amiss, such as missing documentation, the reviewer can make comments, and you can add commits to address any issues. The pull request page becomes a conversation about the changes, where people can make comments on individual commits, lines, or blocks of code to address any issues.

3 Tips

3.1 Just Downloading RAT

Once SSH keys are set up, getting RAT is a one-liner:

```
$ git clone git@github.com:rat-pac/rat-pac.git
```

This puts RAT in a new directory called `rat`. To put it somewhere else, tack the destination directory name onto the end of that line. You can also download RAT over HTTPS (substitute the URL https://github.com/rat-pac/rat-pac), without involving SSH keys at all.

To update with changes from the main repository, run `git pull`.

If you change your mind and want to do development in this directory, you can follow the instructions in Section 2.3.

3.2 Checking Out an Old Revision

When you clone a repository, you have the entire history on your computer, so you can check out any revision, even while on a plane. Run `git checkout` to get to another revision:

```
$ git checkout X
```

where `X` is a *ref*, as explained in Section 2.5. After running this, your directory is magically transformed into the revision you asked for. If this is a RAT repository and you want to run this version, you'll need to re-build:

```
$ ./configure
$ source ./env.sh
$ scons -c
$ scons
```

If you want to keep several different versions of RAT compiled at once, the easiest way to keep them in separate directories.

3.3 Finding a Commit

Because the history of a Git repository doesn't have to be linear, revisions can't be named with a sequential number. You can find a revision ID by:

- Searching the history on GitHub: https://github.com/rat-pac/rat-pac/search.
- Looking at the history on github.com: https://github.com/rat-pac/rat-pac/commits/master
- Looking at the history by running `git log`: This opens in `less`, so you can search.

3.4 Fixing Common Commit Mistakes

If you made a mistake in a commit but haven't pushed yet, there is still time to fix it. This will change the ID of the commit, so don't do this with commits that have been pushed – fix it in a new commit.

You can easily fix a bad author name or email (these should match your GitHub user information):

```
$ git commit --amend --author="Your Name <email@website.com>"
```

Or a mistake in a message:

```
$ git commit --amend -m "Fixed message"
```

Or add a forgotten file:

```
$ git add forgotten_file.cpp
$ git commit --amend
```

3.5 Accidental Changes to a Branch

Inevitably, in your zeal to code, you will start editing files and realize you've forgotten to make a topic branch, so you are editing 'master' or another branch you didn't intend to modify. Do not despair; Git has your back.

```
$ git stash  # "stash" your outstanding changes
$ git checkout -b BRANCH-NAME  # make that topic branch
$ git stash pop  # apply the stashed changes
```

You're back in business.

3.6 Accidental Commits to a Branch

If you accidentally commit changes to the wrong branch (like 'master'), you can still fix it.

```
$ git checkout -b NEW-BRANCH-NAME  # make a topic branch based from the current branch
$ git checkout MESSED-UP-BRANCH-NAME  # go back to the messed up branch
$ git reset --hard HEAD^  # reset state to the previous commit
```

If you wanted to apply the changes to a different, existing branch, see `git help cherry-pick` for a tool to do this.

3.7 When a Pull Request Has Merge Conflicts

When you submit a pull request, you may be asked to merge in the main RAT 'master' branch. Follow the instructions in Section 2.4.2: Git will identify the conflicts for you. Fix them, commit the merge, and push the branch to GitHub to update the pull request.

4 Git and GitHub Resources

A few of the great resources for getting acquainted with Git:

GitHub Bootcamp help.github.com

Git Cheat Sheets help.github.com/git-cheat-sheets

Complete Git command reference gitref.org

`git help` Git is very well-documented – try, e.g. `git help branch`.