Gerrit is Awesome

April 21, 2016 at 12:10 PM | Code

In the past week I've started using Gerrit for all development on SQLAlchemy, including for pull requests, feature branch development, and bug fixes across maintenance branches. I was first introduced to Gerrit through my involvement in the Openstack project, where initially I was completely bewildered by it. Eventually, I figured out roughly enough what was going on to be productive with it and to ultimately prefer it for adding code changes to a project. But making the big leap of actually moving SQLAlchemy and my other key projects over to it required the extra effort of learning Gerrit a little more fundamentally as well as installing and integrating it with SQLAlchemy's existing workflows.

What is Gerrit?

I don't consider myself to be any authority on Gerrit, so my description here is based on my working impression of what it does; the details may not be entirely accurate. Gerrit is a "code review" tool, intended to allow collaboration around code changes targeted at a project. At that level, what Gerrit is doing can be compared to a pull request - allows proposal of a change, shows you how it's different from what's there already, allows comments on the change including against specific code, and then provides workflow to allow the change to be merged into the code repository. For those familiar with pull requests, this seems exactly the same, yet while I've always been very unsatisifed with pull requests, I am completely satisifed with Gerrit's model. So to understand that requires more understanding of Gerrit.

At its core, Gerrit is maintaining a set of git repositories that are basically mirrors of your projects' actual git repositories. Then, it adds some additional, non-mirrored (e.g. only to its local copy) reference paths to these git repositories such that you can push commits to them which represent **Code Reviews**. These commits resemble feature branches in that they are branched off of "master" or another maintenance branch, but the way we work with them is different. The most immediate difference is that unlike a traditional feature branch, the individual "code review" is always a *single commit* containing the entire new feature all at once, whereas a feature branch may consist of any number of commits. In order to allow additional changes and refinements to the "Code Review" as it proceeds, you create a brand new changeset that encompasses the previous changeset completely, plus your new changes. Since there is no longer a linear history in Git between these refinements, Gerrit adds an additional "Change-Id" identifier to the commit message which is how Gerrit keeps track of each change in a code review. So instead of a long-lived feature branch that changes by adding new revisions to the end of it, you have a series of discrete all-at-once commits, each one containing the whole feature at once.

In ASCII art parlance, the traditional Git branching model, also used by pull requests, can be seen like this:

```
rev 1 -> rev 2 -> rev 3 -> rev 4 -> rev 5          -> master
                    |
                    +-> rev 3a -> rev 3b -> rev 3c -> feature branch
```

Above, the feature branch is forked off from the code at some point, and as development continues, a new revision is added to the feature branch. The multiple commits of the feature branch will eventually be merged into master, either using a traditional merge, or by "rebasing" the feature branch onto the top of master.

In Gerrit, the open-ended nature of Git is taken advantage of in an entirely different way, and the mapping of Gerrit to Git might look more like this:

```
rev 1 -> rev 2 -> rev 3 -> rev 4 -> rev 5 -> master
                    |
                    +-> rev 3a
                    |      |
                    +-> rev 3a/3b
                    |      |
                    +-> rev 3a/3b/3c
                           |
                           v
                       Change Id: Iabcdef
```

The above model would be difficult to work with using Git alone; Gerrit maintains the list of changesets conforming to a Change Id in its own database, and provides a full blown GUI and command line API on top so that the history of development of the "Feature" is very clear and easy to work with.

Getting Used to It

The major caveat with this whole approach is that there's a non-trivial conceptual hill to climb in order to use it, particularly if you're not accustomed to "git rebase" (as I wasn't). When we are in our code change that we've pushed as a code review, and we want to revise it and push it up as a new version of that code review, we need to **replace** the old commit with the new one, not add it on as additional history. The most fundamental thing this means is that instead of saying `git commit`, we have to say, `git commit --amend`, meaning, squash our current changes into the most recent commit. The hash of the recent commit is replaced with a brand new one, and the old commit is essentially floating, except for the fact that Gerrit will track it.

When we amend a single commit, we get the same commit message and we basically leave it alone; the commit message represents what the change will be a whole, not the little adjustment we're making here. The commentary about adjustments to our feature occurs in the comment stream on the review itself. With Gerrit's approach, you no longer have commits like, "fix whitespace", "add test", "add documentation" - at the end of the day there will be just one commit message with, "Add Feature XYZ; XYZ does QPR ...". I consider that an advantage, because intermediary commits within a feature branch like "fix whitespace" are really just noise; I don't miss them.

The other thing that has to be dealt with is that Gerrit won't allow you to push up a code review that can't be cleanly merged into the working branch it will be a part of. Again, this is a conceptually odd thing to adjust to but when you start doing it you realize what a great idea it is; when we work with tradtional feature branches and pull requests, there's always that time when we realize we've fallen *so* far behind master, and now we have to merge master into our branch, and fix all the conflicts, so that we can merge back up later without making a huge mess.

With Gerrit's approach, nothing ever gets pushed up that can't be immediately merged, which means if you're targeting a project that has a lot of activity, you'll find yourself having to rebase your code reviews very often - but the key word here is "rebase". Instead of merging master back into our feature branch, we are doing a straight up rebase of our single commit against the state of master; and because there's only one commit to deal with, this is usually a very simple process - we aren't burdened with trying to knit the changesets together in just the right way, or worrying about awkward merge artifacts cluttering up our feature branch forever; we *flatten everything into our single changeset*, and all the clutter of how we merged things together is discarded. As long as you're comfortable with "git rebase", it's a predictable and consistent process. If you are using

Gerrit for any amount of time, you will be *very* comfortable with rebasing :).

Advantages

The advantages to the above Gerrit model are in my opinion huge wins, including:

1. You always have a feature packaged up as a single, clean commit, so that even with "git show", you can see the change represented by the feature all at once in its entirety, with no stream of trivial "work in progress" commits cluttering it up. Because Gerrit maintains its own database of past versions and commentary, the history of how this change was developed is persisted permanently, but outside of your Git repository.

2. You don't clutter up your Git repository with tons of old feature branches. Git of course allows you to delete feature branches, but then you lose all the change history. With Gerrit you get to hold on to a permanent record of how a new feature or change was produced and none of it clutters up your main repository.

3. Any number of developers can collaborate on a single change with no bumps in the process. This is something that is just not practical at all with pull requests; if a user submits a pull request to me, and I'd like to add some changes to it myself, the usual answer is that I need to pull that pull request into my *own* feature branch somewhere, and change it there - totally separated from where the original pull request is. Or, the developer of that pull request can opt to give my account write access to their repository; however in practice, I've never seen anyone do this, and it also means I'm awkwardly working on my own project via someone else's account. This is the most continuously frustrating thing about pull requests and is a key reason I'm so much happier with Gerrit. With pull requests I often found myself having to painstakingly describe exactly how some part of code should look in order to get the submitter to do it in their branch, so that I wouldn't have to "take it over" and chase them away forever. With a Gerrit code review I can just push up a modification to their change, and the contributor can jump right back on with no bump in continuity. When I've made a lot of changes to someone's change I add "Co-Authored By: " with my name to the commit message so that a viewer can see it was a two-person job.

4. The "change as a single commit" model means we no longer have to deal with pull requests that consist of two dozen changesets, re-merges of master, weird merge artifacts like where you see zillions of merged commits interspersed with the ones you care about, and so on. When I go to "git merge" someone's pull request, I usually need to go through the effort to manually squash it and add my own changelog notes, which means that the Github or Bitbucket UX has no record at all of me "merging" their pull request, and most horribly on Bitbucket I have to explicitly mark the PR as "DECLINED", when meanwhile I just wanted to squash it. Gerrit on the other hand is built for rebasing and squashing from the ground up, and represents an accurate record of **exactly** what was merged, with no chance that I went off and modified someone's pull request locally after the fact in a short-lived feature branch. **Everything** on a code contribution from start to final merge to master is visible exactly within the Gerrit UX, with complete accuracy and permanence.

5. The workflow model of Gerrit is far richer than that of a pull request and is also fully customizable. Different roles, such as that of developers, contributors, and automated CI systems, can each get their own workflow flags each with different meanings. Typically, a core developer of the project needs to assign a "+2 Approved" flag, the CI systems in play need to assign "+1 Verified" and in Openstack and on my own system there's additionally a "+1 Ready to Merge" flag; when all those are up, you press "submit" and the code review is pushed to the target branch, and its done. With Bitbucket and Gerrit pull requests, I *never* pushed the button, as I was always having to add changelog notes, make fixes, squash it, fixing merge conflicts, all of which I can now push straight into that same code review under

one consistent interface.

6. Integration with Jenkins is way better for Gerrit than it is with Github, and for Bitbucket I've not even been able to find any Jenkins integration. Once I pull a code change into Gerrit I can easily grant the contributor access to continue working on it, or disallow changes, so that I can on a per-user basis decide who gets to push new changes that automatically kick off builds and not have to worry that someone will submit a malicious pull request when I'm not looking. With the Jenkins Github pull request plugin, it was simply not an option to enable automatic CI unless I were using something like Travis, which currently I'm not.

Overall, with Gerrit I now have a single, consistent way to do *all* new code; all of my feature branches, all of my random .patch files sitting in my home directory, pull requests from github or bitbucket; all of it goes straight into Gerrit where I have them all under one interface and where changes, history, code review, workflow, and comments on them are permanent, without creating any branch junk in my main git repo!

Gerrit has a ton more features as well, including an extremely detailed permissioning model which you can map to specific paths in each git repo; a Lucene-powered search feature, a replication engine so that changes to the local git repo can be pushed anywhere else; it is itself a full blown http and ssh server for the git repositories as well as for its command-line and GUI interfaces, and generally has all-around industrial style features.

Where Gerrit is falling short is that on the "self-install" side, it is definitely a little obtuse and unforgiving about things. It was not that easy to set up, and you'll likely have to spend a lot of time reading stack traces in the logs to figure out various issues - documentation, links to packages, and other online help is a little scattered and sometimes incomplete, especially for the plugins. It's not very user friendly on the adminstration side yet. But hosting it yourself is still the way to go, so that when you hit a view like "all", you see just your projects and not a huge list of other people's projects.

Implementation

The move to self-hosting services again is kind of a pendulum swing for me; a couple of years ago I was entirely gleeful to move my issue tracking off of Trac and onto Bitbucket, but i think the reason for that was mostly due to Trac's inability to limit spam accounts as well as a burdensome performance model and very little upstream development. With Gerrit, I'm self-hosting again, however all authorization is pushed up to Github with OAuth, so that I'm not dealing with fake user accounts and so far not any spammers either.

I'm using the Gerrit Oauth plugin to provide Github logins, and this plugin also includes support for Google Oauth which I haven't turned on yet, and Bitbucket OAuth which I couldn't get working.

For integration with community-supplied code submissions, I am still using pull requests as the primary entrypoint into the system. Github and Bitbucket require that you leave the "pull request" button on in any case, so people are naturally going to send you code on both of them no matter what you say, so we might as well use them. The pull request allows me to have a quick preview of what we're dealing with, and I then import it into Gerrit using a semi-manual process. To achieve this consistently, I've created the prtogerrit script which communicates with both the Github and Bitbucket APIs to pull in any target pull request, squash it, push it into Gerrit as a new review, and add comments to the pull request referring the contributor to our Gerrit registration page. Pull requests submitted through Bitbucket still necessarily get that big ugly "DECLINED" status, however it is now consistent for all BB pull requests and the messaging is clear that the work is continued on Gerrit.

Sum Up

Since I've been using Gerrit, I've been surprised at how much more productive I became; being able to see every code change being worked on by anyone all under one interface is very freeing, and the workflow that lets me have the entire change from implementation to tests to changelog and migration notes in one clean, mergable commit, fully run through multiple Jenkins CI jobs before it gets merged has made me totally comfortable pushing a big red "merge" button at the end, which was never the case with pull requests and other ad-hoc patchfiles and feature branches.