BACK TO LIST

Git or SVN? How Nuance Healthcare chose a Git branching model?

Matt Shelton

This is a guest post from Matt Shelton at Nuance Healthcare. This is the first post in a series about his team moving from Subversion to Git, why they did it, and what we encountered along the way. Matt is also speaking on this topic at Atlassian Summit 2015. This series will feature everything he couldn't say in his 30 minute talk, with more context.

Background

My team is in the healthcare division at Nuance. We're geographically-distributed between a couple of offices and homes on the East Cast of the US, and in an office in Pune. We develop Java web services to deliver NLP[1] solutions to the healthcare market.

BACK TO LIST

For the most part, our service consumers are other healthcare software companies (including ourselves) such as EHR vendors and healthcare analytics companies. We do directly sell some products to hospitals, and the end-users of the applications range from physicians to medical billing staff. "Normal" people like you and me don't ever touch the software my team builds.

Our team has been around the block a few times with Application Lifecycle Management product combinations. We started life with a mix of Rally Enterprise and Seapine TestTrack Pro, did about 14 months of hard labor with Rational Team Concert, and eventually migrated fully to the Atlassian stack (Jira, Confluence, Bamboo, Crucible, Bitbucket and Hipchat). Historically we used Subversion 1.4/1.5 an our SCM with a quasi-normal trunk/branches/tags structure. We have been using maven since forever manage our build projects and dependencies, and switched from Jenkins to Bamboo for continuous integration (CI) a while ago in order to make use of tighter integrations with Jira and its flexible build and deploy agent capabilities. Everything we use (now) is behind the firewall for reasons[2].

Git or SVN?

We support roughly ten individual products across four product families, and the owners of these products are *always* battling for prioritization and timing. It's nice to have our work be in high demand, and this is by no means a *complaint*, but it also necessitates cutting releases at a weird cadence and needing to change directions in the middle of a sprint[3].

Our development process really felt prohibitive at times. There was a conversation that my team was having on a regular basis that went

something like this:

> **Me**: We need to release 1.8.0 to QA now for regression testing so that Customer foo can go to beta next week. **Dev**: I'm still working on ABC-123 which is in trunk. It's not done yet. **Me**: Foo doesn't need ABC-123. We could put it in the next release. **Dev**: But I've been working on it for weeks. There's no clear spot to branch from to cut a release. **Me**: Well, you'll need to pull out all of your changes by hand. You have about two hours or QA can't finish in time.

BACK TO LIST

I know, I sound like a jerk! I never meant to be, and of course I'm exaggerating a bit to make a point, but we really did have to figure out how to get code that was in one place out of that place temporarily so that we could cut a release, and then *put it right back*for the next release[4]. And this happened all the time.

Now, I know some of you are thinking "Subversion supports branches, Matt...". It absolutely does, and we used them on occasion with SVN 1.4 and 1.5. Branching is a fine operation in SVN; *merging*can be a pain in the ass. As SVN has matured, it has gotten better, for sure. But we knew there were better options out there *for us*, so when the question of SCN or git arose, we set out to get Git.

> A side note: We briefly looked at the latest SVN (1.8 at the time) to see if it was strong enough to solve our problems, but weren't completely satisfied. One of our peer groups has a large Perforce setup and it had a lot of what we needed, but I simply couldn't stomach the licensing costs. We also looked at Mercurial for a moment, but in the end, the existing team's exposure to Git was enough to tell us that it was the right direction.

BACK TO LIST

I won't sugar-coat this: Atlassian's tools really favor teams who use git. Other SCMs work fine; our SVN integration was *sufficient* in that it linked us to where a given user story's changes were made. The integration capabilities for teams who use Bitbucket Server[5] instead, however, are both stronger and more natural-feeling in the Jira Software interface and development experience - ditto with Bamboo.

Knowing this, and having seen some very stellar demos at Summit 2013, I strongly encouraged the team to go for it. Nobody objected, and we already had the licenses in place to make change.

Choosing a Git Branching Model

After deciding to make this change, the first challenge we had was deciding what Git branching model to implement for our team. Atlassian's Git microsite as well as this great presentation from Summit 2013 explain in greater detail what a branching model is. The short version is that it describes how you will use branches in git to power your development workflow.

In SVN, we had a model for branching I'll call "make one when you realize you - OMG! - need one":

- The newest code is in `trunk`. Releases from trunk will be numbered `A.B.0-{build}`.
- If a fix is required to a trunk-based release (e.g. we have a bug in 1.2.0-64), a branch is created and from there we will release `A.B.C-{build}` releases, where `C` increments after every release that goes out the door. These branches may never exist for a given `A.B` and we could even have more than one.
- We also tag every release in a tags directory.

BACK TO LIST

An Aside About Versions Many years ago, when I was just cutting my teeth on managing a development team, our release engineer had a system of versioning that was... how shall I say?... *really unintuitive*. Essentially, every release was a patch on the previous one (A.B.n), with no respect for the place from which the patch originated. Figuring out where something came from and, in almost all cases, the *release order*, required you to look at `svn log`. We printed the tree on a wall for reference. In addition, our public-facing release numbers tend to be things like 3.0, 3.1, 3.5, 4.0, or essentially something a customer might expect. Remember, though that my team builds *web services* not a boxed product. Our APIs are a contract. A few years ago I made the executive that my team's builds, and therefore its releases, would adhere to Semantic Versioning rules. I've' had to stand my ground a few times with upper management, but now it is understood why the rules are what they are, and we haven't looked back. Partners appreciate that sort of clarity.

I mentioned a problem earlier wherein we'd be working on a release (let's say `1.2.0`) and we'd have a feature still in progress as we approached a release date. We would need to pull that code out, cut our release, branch to `branches/1.2.1` and then merge that code back in, hoping nobody had a hard drive crash in the meantime[6].

Removing a whole feature by itself from a shared trunk is a pain. Everyone hated life when they had to do that. `svn blame` can be useful, as can a strong diff tool, but it’s still annoying to work with. I often took it personally, feeling that my bad planning had led to us not having all of our ducks in a row before it was time to be done with a release[7]. My team dealt with this for long enough.

Sometimes we'd over-correct to avoid the pain and would ask developers to sit on their hands for a couple of days (a virtual code

freeze, if you will), just so we didn't pollute trunk before a release.

So we knew we needed, at least, feature branches. There's a simple Git branching model that is applicable: a master branch for what's in prod, and using feature branches for every feature, bug, etc. Team's have to manage merge order to ensure that what ships out in master is what is supposed to ship out for the release. This is, essentially, the same thing we had before, with some better feature isolation, but we wanted freedom with our power.

BACK TO LIST

In our environment, we often need to keep a few versions in production, and may need to fix defects in a release that is 2-3 minor revisions older than what we are working on right now. So, in addition to feature branches, we also needed some sort of release branch or similar that would let us fix issues from previous releases. The Atlassian Bitbucket Server team does this. They make fixes in long-running support branches, and then merge them up the branch stream so that a fix makes it in to all of the support streams.

Their model looked really good, and we ran a few prototype interactions with this model to see if it would suit our needs. The "killer app" for them is their rolling merge of a fix up to their develop branch. While we liked this concept, every time we tried it, we ran into one issue or another with our maven dependencies. Also, as a rule, we couldn't guarantee we wanted a straight merge of the work from one version into another. In some cases we needed to implement the same fix in slightly different ways between versions, so a direct merge wasn't possible.

A few of the members of the team strongly favored a variation of this model known as "git-flow". Git-flow is a set of branch naming conventions and merge guidelines, authored by Vincent Driessen. This felt very natural to the team, and we liked the structure since it eliminated many of the questions around "what do I do when we need to do x?". The answers were generally very obvious. Rather than

explaining what git-flow is, you can read more about it in Atlassian's tutorial.

BACK TO LIST

The only gap left for us with git-flow was what to do about those long-running releases in production. Since master keeps moving forward, we couldn't use the git-flow hotfix workflow for a bug fix from a previous release. On the other hand, we didn't *always* want a support branch.

Most of the time a hotfix, only patching the latest release in production, should be sufficient; support is only there when we need to go back further, or when we need to maintain compatibility for one reason or another. That latter use case we dissected further and came up with criteria for choosing to use a support branch rather than a hotfix and minor version upgrade:

1. This code cannot be trivially merged back into develop.
2. The partner/customer cannot handle an interface change that comes with the latest release.
3. There is an internal dependency which cannot be changed.[8]

Both git-flow extension packages[9] provide support for the support branch concept, which isn't part of the original draft of git-flow, but has become popular enough to warrant inclusion.

Git-flow offered a workflow we liked, with the tooling support we needed. In the next post I'll go into what happened when we actually tried using it in a POC project we used to represent our development process. It was... a learning experience!

[1]: Natural Language Processing. WE CAN READ YOUR THOUGHTS. (No. Not really.)

[2]: There is a *lot* that is attractive about Atlassian's cloud offerings, but we need to keep our fingers wrapped tightly around our servers and data for the time being. While we don't personally need to do

much with PHI data, our software *does* and it's important to keep it as secure as possible.

[3]: Shhhh... don't tell Ken Schwaber.

BACK TO LIST

[4]: Which might have only been a few days later anyway.

[5]: Formerly known as Stash. Hello, Atlassian Fall Rebranding!

[6]: I know we could always pull it out of the previous commit. I was kidding.

[7]: This wasn't usually the case - generally it was because someone else's timeframe moved up and we had to react quickly.

[8]: This is one of those things I can't get into on my own blog. Just trust me. "Reasons".

[9]: The original package by Vincent Driessen isn't being maintained any longer. A new fork , however, is regularly updated.