Software is too expensive to build cheaply….

Robert's Rambling Ruminations Regarding Reality...

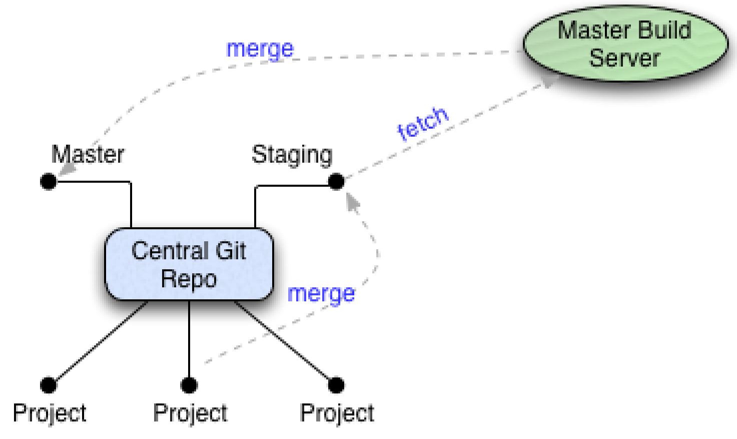
Making Parallel Branches meet regularly with Git and Jenkins

If you've been following my tweets recently, then you'll know that we've recently converted the majority of our projects at work from Subversion to Git for our source control. We didn't just do this because we wanted to play with a new shiny toy, but because we hope to achieve a new way of working. That's what I want to describe here.

Let me set some background. At work, we've got three development teams (in our section, anyway). These teams look after three biggish externally facing applications, and about a dozen internal applications and systems. There's also about half a dozen important shared libraries – so call it about 20 different modules. Of these, about six are what I would describe as 'high contention' – most projects that we do want to change several of these. This means that we do parallel development on these modules. We don't want to, and we have done a lot to reduce the extent of the overlap, but it's a fact of life for us at this time.

Parallel development bites, in so many ways. One of the biggest is that you get semantic conflict, especially in regards to refactoring and other forms of improvement. This makes merges difficult and time consuming – I've estimated that we have, at various times, lost upwards of 3 days a month to merges per senior developer. That's nearly 15% of the time of our most experienced developers lost – and that's a cost that was not sustainable. Git – combined with a change in workflow – held out a hope of a solution.

So that's the background. Here's what we're doing.



We have a central 'blessed' repository server – this holds the definitive version of the git repositories, and it gets backed, exported to other tools (such as FishEye), and so forth. We then have a master Jenkins build server which checks out changes from the repositories – on a special '*staging*' branch – and builds them. On a successful build, the *staging* branch is fast-foward-merged to the *master* branch.

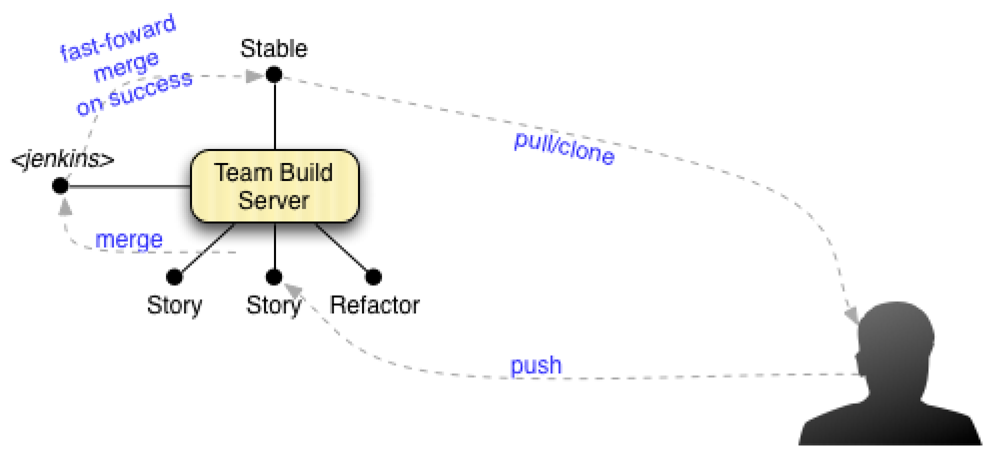
The point of the *staging* branch is that's where projects do merges to. In an ideal world, merges will always be simple and painless (due to the steps I'll describe later), but let's be realistic – there will still be some cost, and there will still be errors. By using a *staging* branch to do merges on, we keep the *master* branch clean so people can make new branches.

Nobody is meant to commit to the *master* branch (except for the build server itself), and we will eventually enforce that using pre-commit hooks (managed via gitolite). Keeping the master branch clean is intended to be an inviolate rule.

The master build server is fairly idle – after all, it only gets triggered on merges, which are associated with project deliveries. This is not for day-to-day use. For that, we've got team build servers.

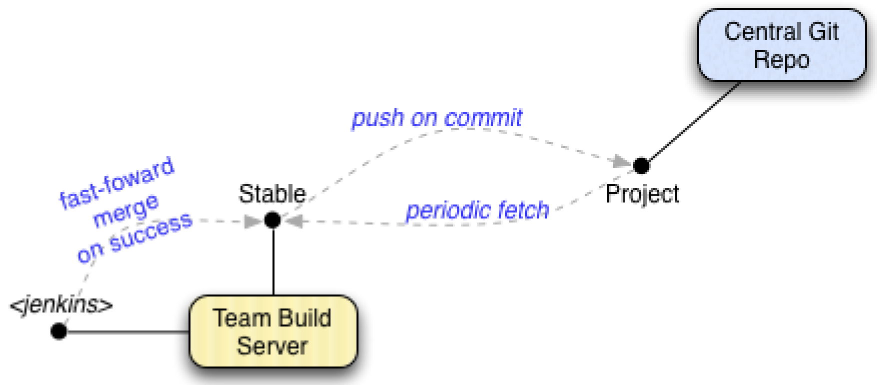
This picture shows how our team build servers work. The build servers are also git repositories – they are clones of the central repositories. The *stable* branch here is a tracking branch – tracking the corresponding project branch back on the centralised server.

Developers use the team build server as their main remote repository. When they start a new story, they do a *git fetch*, then they *git branch –track <story_name> <buildserver>/stable* – creating a FeatureBranch from the current *stable* (that is,



passing builds) branch for the project. This feature branch gets pushed back to the build server whenever the developer is ready to do so, or about as often as they would have done commits when using Subversion. (Hopefully they're doing more commits locally, to get more power out of using Git).

The build server uses the Git plugin for Jenkins to monitor all the branches on the local repository. Whenever a developer pushes to the repository, Jenkins will see the change and try to merge it into the *stable* branch. If the build passes, the merge is committed. If it doesn't pass, the FeatureBranch doesn't get merged – and it will stay unmerged until another change is made against it. This means that the *stable* branch is always clean as well. Again, nobody is meant to commit against the *stable* branch except for the build server (and, again, this will be enforced via pre-commit hooks).

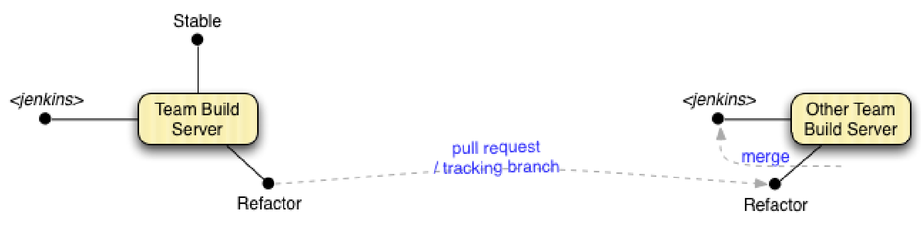


Whenever a merge is made to the stable branch, a post-commit hook pushes the change up to the central git repo – on the project branch, of course. We do this because our build servers are transient – they're VMs, and we periodically blow them away, partly because we update the VM templates occasionally, but mostly because it's an easy way to manage the disk space. Getting the code up to the project branch is important for backup purposes.

(Nobody is meant to commit directly to a project branch either – but as a safety net, we do periodic fetches – and branch resets of *stable* – from the central server, all managed by *cron*)

An interesting consequence of this that every branch on the central server is clean – if it didn't pass the build, then it wouldn't have merged to the *stable* branch, and wouldn't have been pushed to the central repository.

So that's normal stories. However, the real power here are refactoring branches. There's a simple rule for this: if a story is hard, you need to refactor and clean up – on a refactoring branch – until the story is going to be easy. Then you do the story. This separates the work that changes the semantics of the system from the work that extends or enhances it.



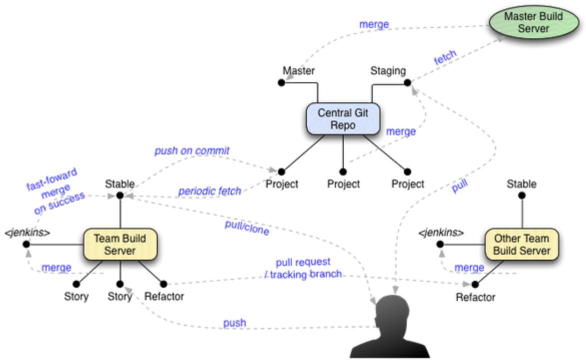
The real power here is that the refactoring branches can be more readily merged across to other branches. After a refactoring branch is (reasonably) robust, it can be added as a mirror branch on the build server of another team. This will automatically get pulled into the development line of the other team (via the same scripts that keep the servers up-to-date with the central repositories). Alternatively, it can be added as a normal remote tracking branch and merges can be managed by hand.

(Thanks to our virtualised build infrastructure, we could even spin up a new build server/repository just for the refactoring branch if we want – lots of options)

Under Subversion, this would have been a nightmare. Even with merge tracking, this would have produced conflicts like crazy. Because Git tracks the commits themselves, though, merges back to the *staging/master* line aren't made any more complex. This gives us a very strong incentive to make sure that these refactoring branches are integrated across the parallel branches continually – we all really want to avoid the semantic conflict!

(To make this work best, the refactoring branches should actually be branched off the central repository *master* branch – but that's easy enough to do anyway)

And here's what this looks like as a big picture – click to embiggen:



So that's the reason we've moved to Git – so we can support parallel branches, using Feature Branches for day to day work, with cross-team refactoring branches to avoid semantic conflict – and how we've done it – using Jenkins and a handful of support scripts that automate the job and repository management. We're still in early days with this, of course – it's only been two week since we started – but the early signs are very promising.

Share this:

Tweet Share 0 SHARE Print More

Like

One blogger likes this.

Related

[Git, Feature Branches, and Jenkins – or how I learned to stop worrying about broken builds]

In "Agile Development"

[… and sometimes they don't.]

In "Java"

Setting up a Jenkins Server with Docker - Adventures in Learning

In "Agile Development"

Author: Robert Watkins

My name is Robert Watkins. I am a software developer and have been for over 20 years now. I currently work for people, but my opinions here are in no way endorsed by them (which is cool; their opinions aren't endorsed by me either). My main professional interests are in Java development, using Agile methods, with a historical focus on building web based applications. I'm also a Mac-fan and love my iPhone, which I'm currently learning how to code for. I live and work in Brisbane, Australia, but I grew up in the Northern Territory, and still find Brisbane too cold (after 22 years here). I'm married, with two children and one cat. My politics are socialist in tendency, my religious affiliation is atheist (aka "none of the above"), my attitude is condescending and my moral standing is lying down.

View all posts by Robert Watkins

Robert Watkins / 16 September, 2011 / Agile Development / git, jenkins, maven, rube goldberg, version control

7 thoughts on "Making Parallel Branches meet regularly with Git and Jenkins"

Pingback: Git, Feature Branches, and Jenkins – or how I learned to stop worrying about broken builds – Software is too expensive to build cheaply…

Doug Beatty

6 December, 2011 at 14:14

This is an interesting solution.

So it looks like you have been at it about 3 months now. How is it working? Any updates changes/tweeks?

Do you have any further references on this subject?

Thanks,

Jon Lachelt

22 May, 2012 at 09:00

We're planning to implement something like this, but we don't want the build server to automatically merge a feature into the stable branch just because the build passed the tests. The developer may not be finished with the feature, or we may need product manager approval before we decide to merge the feature into the stable branch. So we want some additional action to trigger the merge (but still want it to happen only if the build/test were successful).

Robert Watkins

28 May, 2012 at 22:07

In that case, I'd suggest merging it into some sort of candidate-for-stable branch. The biggest problem you'll have with long-lived branches is that they will get harder and harder to merge and integrate together; keeping a candidate-for-stable branch around will make it easier to identify that.

Pingback: Building Dependent Maven Projects in Bamboo | Software is too expensive to build cheaply….

Paŭlo Ebermann

10 December, 2013 at 23:22

The dark gray background doesn't work well together with these transparent images (at least in Chromium). Maybe the images should not be transparent, or need differently-colored lines and text?

Robert Watkins

10 December, 2013 at 23:25

I had a white-background theme when I originally posted this article; I change periodically.