Super-powered CI with Git

SARAH GOFF-DUPONT • DEV TOOLS MARKETING • ATLASSIAN • @DEVTOOLSUPERFAN

Good morning! My name is Sarah and I'm on the DevTools marketing team at Atlassian. And today I'll be sharing with you some of the things that our teams have learned about doing CI and using Git. This is kind of an intermediate-level talk, so if you're looking for really advanced info, you might be better off finding another session.

That said, it's really exciting to see so many people here today, taking the time to learn more about continuous integration and Git – and how they can be super powerful together.

Agenda

WHY GIT?

CI-FRIENDLY REPOS

CI ON FEATURE BRANCHES

GIT HOOKS FOR CI

I'll start off today by talking about why Git is particularly advantageous for teams. I'll also go through the branching workflow we use, which is really effective with CI.

Then I'll go over some tips around optimizing your CI system for Git.

And last, I'll introduce some Git hooks that are designed to make your CI system stronger.

Why Git?

WHY GIT?

CI-FRIENDLY REPOS

CI ON FEATURE BRANCHES

GIT HOOKS FOR CI

We know the benefits of making software as a team: you get different ways of thinking, different backgrounds and experiences… When you bring those to bear on whatever problem you're trying to solve, you end up with better software. It's more maintainable, higher quality, and ultimately serves the user better.

But we also know that developing as a team can be messy!

You're trying to understand what pieces everyone is working on, trying to make sure those pieces don't conflict, trying to find defects before your customers do, trying to future-proof your code, and trying to keep everyone connected with the project up to date on how things are progressing.

Well, turns out that Git is very conducive to working in a team setting.

Divide and conquer

First off, Git makes it easy to divide up pieces of a project amongst your team and isolate WIP so developers don't step on each other's work or block a release by putting broken code onto the main branch. Developers work on development branches (often called a feature branch) until the issue or feature has been proven “safe” to merge into the team’s shared branches.

Clearly, SVN and other traditional version control systems let you branch too. But let’s side-step for a moment and meet branching’s evil twin: the merge.

Traditional VCSs like SVN are simply not that great at tracking versions of files that live on different branches, and when it comes time to merge, SVN has to stop and ask for directions a lot. “Do you want this line or that line in the merged version?”

The fact that so much human interaction is required during merges drives teams to institute code freezes so whomever is performing the merge doesn’t get disrupted by new changes coming in on one of the branches. And code freezes are expensive: it’s pretty un-productive time.

So in an effort to minimize code freezes, teams often discourage branching and put rather heavy-handed administrative controls around who can create them. So you end up with everyone putting changes directly on trunk, which then becomes very unstable and difficult to release from or even just make sense of why tests are failing.

But not with Git.

Git is really good at tracking changes to different versions of files that live on different branches, and it always knows what the common ancestor of that file looked like. So it basically has a built-in GPS that lets it navigate merges without having to stop and ask you for directions all the time.

And this makes merging a pretty trivial operation in Git

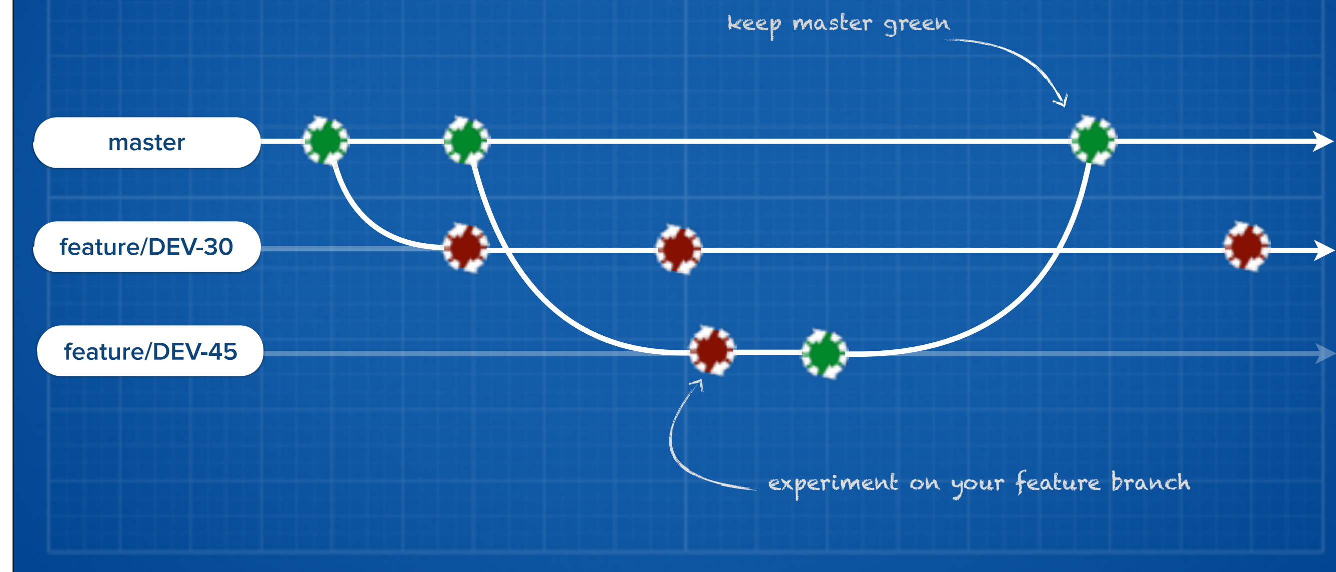
The upshot is that, you are free to exploit the power of branch-and-merge workflows in a way that – practically speaking – you just wouldn't with SVN.

Every team at Atlassian is now using Git, and we are bullish on feature branches. In fact, we've taken to creating a branch for each issue we work on – be it bug, user story, or technical task.

Just in the JIRA code base alone, we have 775 branches. Keep in mind that JIRA is composed of multiple repositories, but still: we're serious about this branching thing.

It might sound like a ton of overhead, but it's actually not – with a good branching workflow, you can go pretty nuts and not feel pain from it. And it's helping us deliver faster than ever before.

By keeping work in progress isolated on feature branches, we can keep master in a clean and releasable state.



The basic workflow we use looks like this. You may choose to make branches at the feature level, rather than for each issue, but the flow looks about the same. We really believe in the branch-per-issue model, and our tools do all manner of magical things around that - feel free to chat me up about that later.

Feature branches are made from master, then when all the tests are clean, the branch is merged back up, and the work can be released.

btw, the little circles have arrows on them, indicating CI

The point I really want to make here is that Git makes it really easy to keep unproven code off of master (or other protected branches), and CI makes it really easy to prove out your code while it's still isolated on a feature branch.

CI-friendly Repos

WHY GIT?

CI-FRIENDLY REPOS

CI ON FEATURE BRANCHES

GIT HOOKS FOR CI

With that in mind, let’s get down to the CI stuff.

Starting with your repo, which is where it all begins.

Avoid tracking **large files.**

One of the things you hear about Git is that you should avoid putting large files into your repo – binaries, media files, archived artifacts, etc.

Once you add a file to your repo, it will always be there in the repo's history – which means every time the repo is cloned, that huge heavy file will be cloned with it.

Keeping large files out is especially important if you are doing CI.

Because each time you build, your CI server has to clone your repo into the working build directory. And if your repo is bloated with a bunch of huge artifacts, it slows that process down.

Just archive it

But what if your build depends on binaries from other projects or large artifacts? That’s a very common situation, and probably always will be, so how can we handle it effectively?

A storage system like Artifactory or Nexus or Archiva can help for artifacts that are generated by your team or the teams around you. The files you need can be pulled in at the beginning of your build – just like the 3rd-party libraries you pull in via Maven or Gradle.

You may be thinking “Oh, I’ll just sync them to the build server each night so I only have to transfer them across disk at build time”

Even though a disc transfer is much faster than network transfer, I actually recommend against doing this – especially if the artifacts change frequently. You’ll end up building with stale versions of them in the builds that fall in between your nightly syncs. Plus, developers need these files for builds on their local workstations anyway. So overall, the cleanest thing to do is to just make artifact download part of the build.

Shallow clones for the win

Next, let’s talk about cloning.

Each time a build runs, your build server clones your repo into the current working directory. As I mentioned before, when Git clones a repo, it clones the repo’s ENTIRE HISTORY by default. So over time, this operation will naturally take longer and longer.

Unless… you use shallow clones for CI.

With shallow clones, only the current snapshot of your repo will be pulled down. So it can be quite useful for reducing build times.

But let’s say your build requires the full repo history – if, for example, it’s a release/official build that adds a tag or updates the version in your POM, or you’re merging two branches with each build (which we’ll talk about more in a bit).

Earlier versions of Git require the entire repo history to be present in order to push changes. As of 1.9, simple changes to files can be pushed without the entire history present, but merging still requires full history because Git needs to look back and find the common ancestor of the two branches – obviously that’s going to be a problem if your build uses shallow cloning.

So you can cache the repo instead, which also makes the cloning operation much, much faster.

And where do you cache it?

…on your agents.

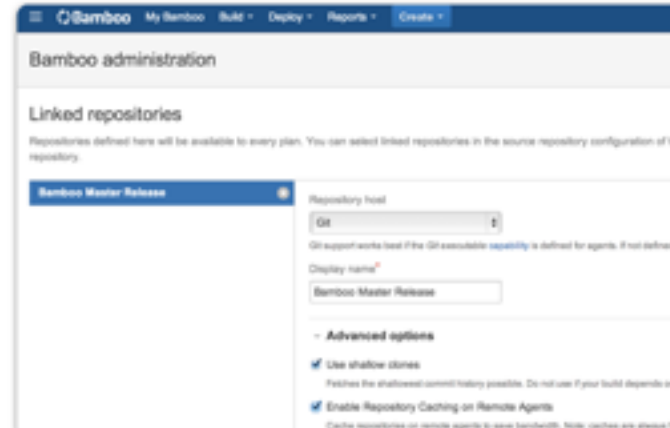
Note that repo caching only benefits you if you are using persistent build agents. If you create and destroy build agents on EC2 or another cloud provider, repo caching won't matter because you'll be working with an empty build directory and will have to pull down a full copy of the repo every time anyway.

Shallow cloning and repo caching are slick ways to manage load on your CI system and reduce build times to boot.

Built into

Atlassian

Bamboo



You can do shallow cloning and repo caching with any CI server – you just may have to perform some gymnastics to get it set up.

If you’re using Bamboo, or are thinking of trying it, setting it up is as easy as checking a couple of boxes.

CI on Feature Branches

WHY GIT?

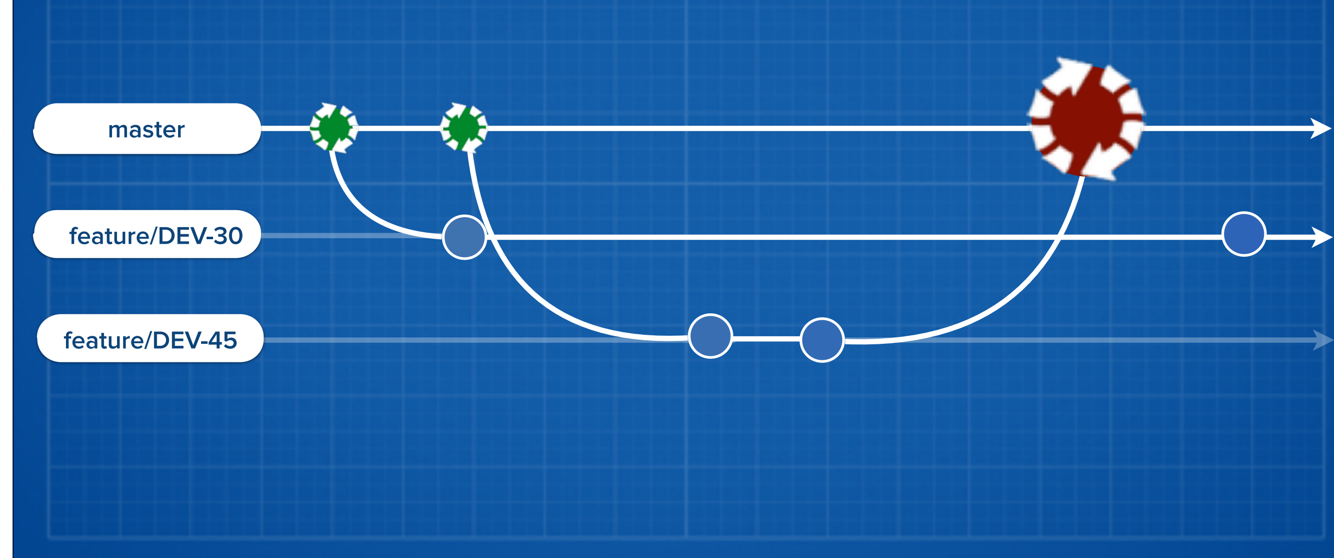
CI-FRIENDLY REPOS

CI ON FEATURE BRANCHES

GIT HOOKS FOR CI

Whether you make a branch for each issue, or a branch for each feature, you end up with a lot of active branches in your repo.

And as people who take our craft seriously, we want those branches to be as clean as possible.



If you're already working with Git, you know that dev branches are pretty short-lived.

Nonetheless, it's really important to run CI on all dev branches. (CLICK)

Otherwise, if the first time you integrate is when you merge your work to master, you'll be hit with all kinds of nasty surprises.

Master is all unreleasable and unsuitable to make a new feature branch from, and that pretty much defeats the purpose of using a branch-and-merge workflow in the first place.

Clone master's CI **configs**

How do you set this up? The old-fashioned way is to make clones of the build for master and point it at your dev branch.

But that's a drag.

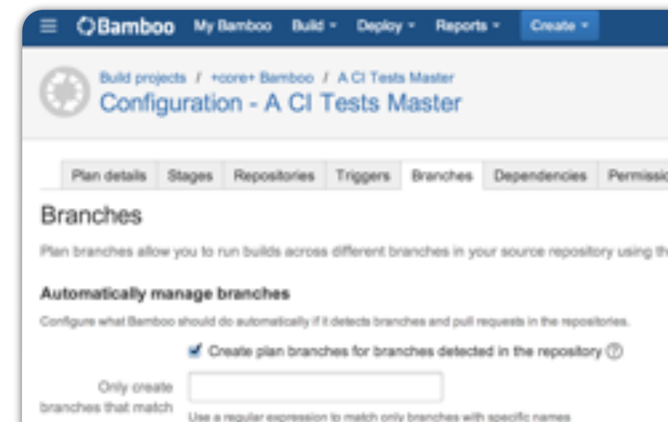
You’ll forget to set up the branch build

We’re all human, and between the tediousness of cloning all those configs, and our own forgetfulness, getting CI onto our branches the old fashioned way is pretty painful.

The new-fashioned way is to make your CI server do it for you.

If you’re still using Jenkins, there’s a plugin called build-per-branch that will discover new branches and apply your CI scheme to them.

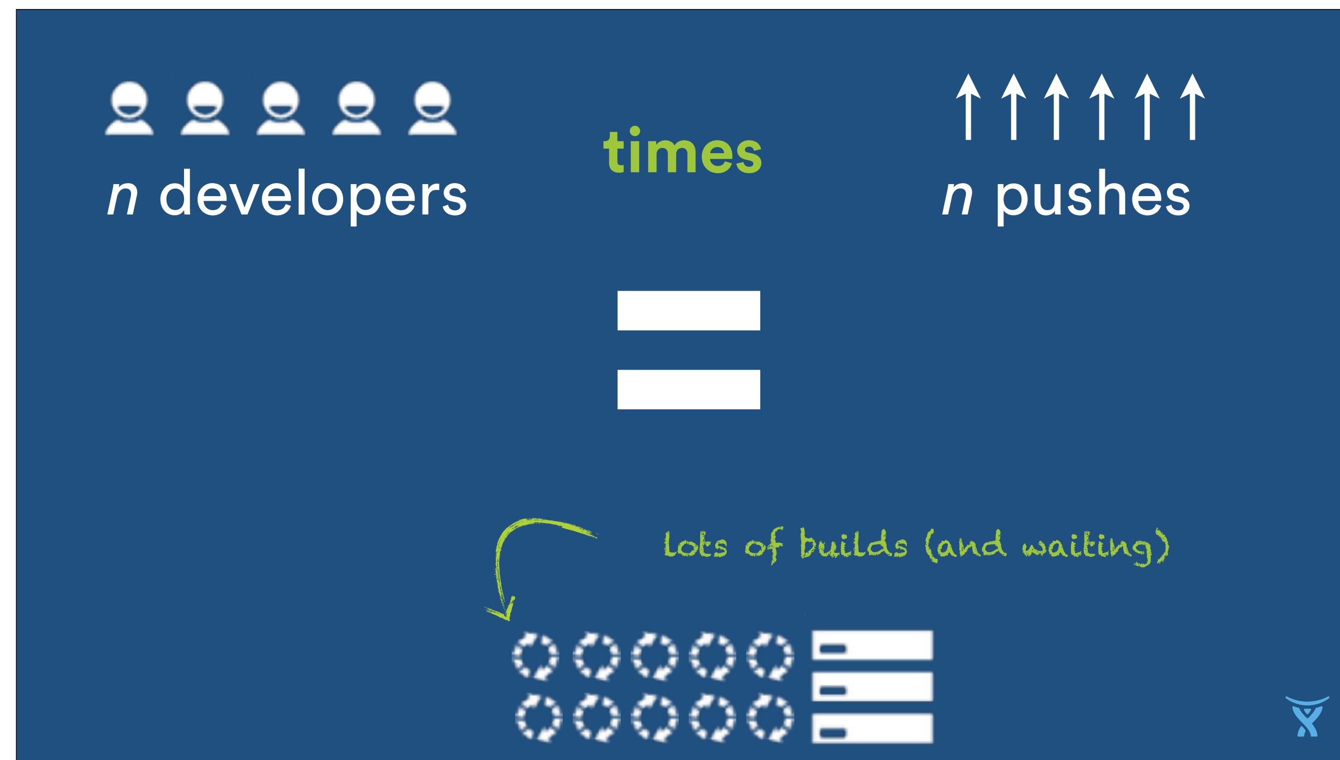
Built into



That plugin is actually a less-robust copy-cat of the branch builds feature we added to Bamboo a couple years ago.

Either way, the point is to spend less time configuring builds, and more time coding.

We discovered just one little problem with all this branch build stuff.



We have almost 500 developers. (CLICK) And they each push changes to their branches several times a day. (CLICK)

If each commit is getting built, that's a lot of builds. (CLICK)

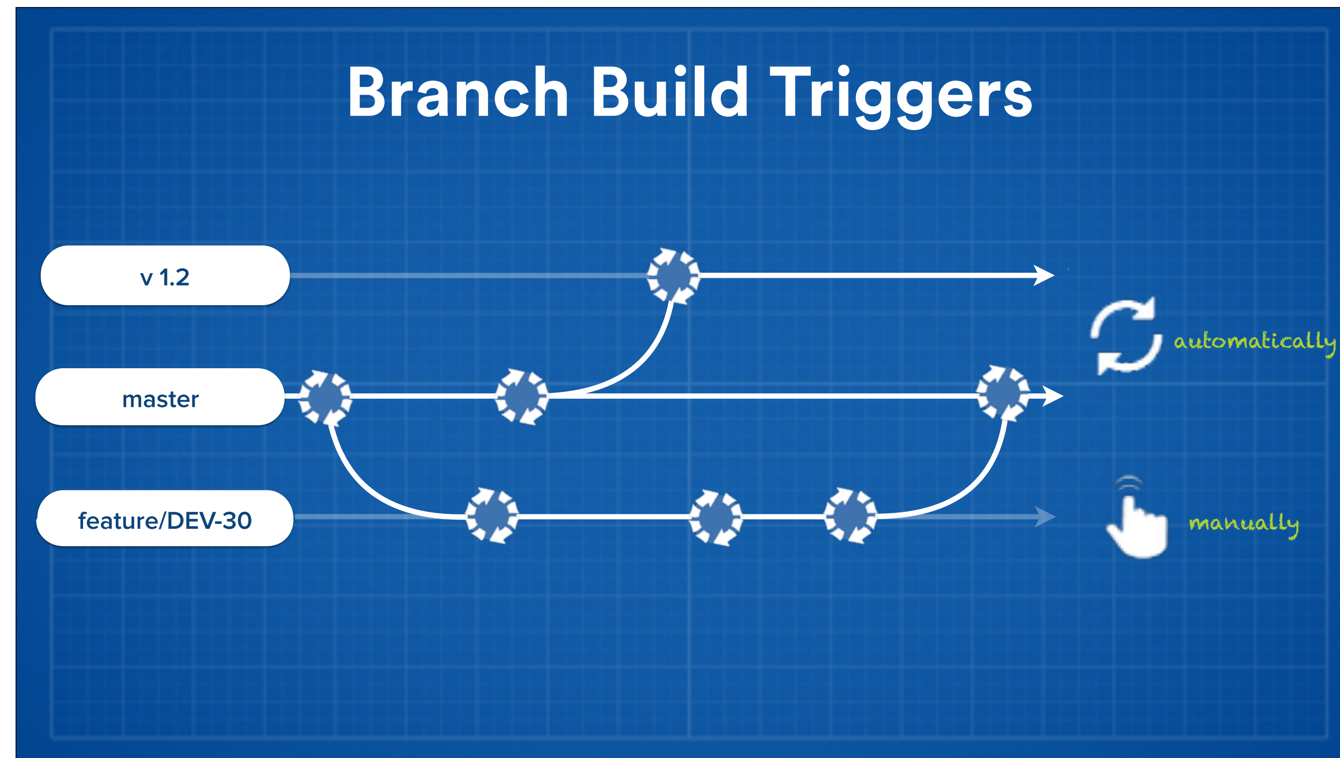
And unless you can scale your build server with infinite build agents, your build queue is going to get backed up.

Take for example, just one of our internal Bamboo servers. It houses 935 different build plans. Not all of them are active, but still. We've plugged ~141 build agents into this server, and used best practices like artifact passing and test parallelization to make each build as efficient as possible.

And still: building each commit was clogging up the works.

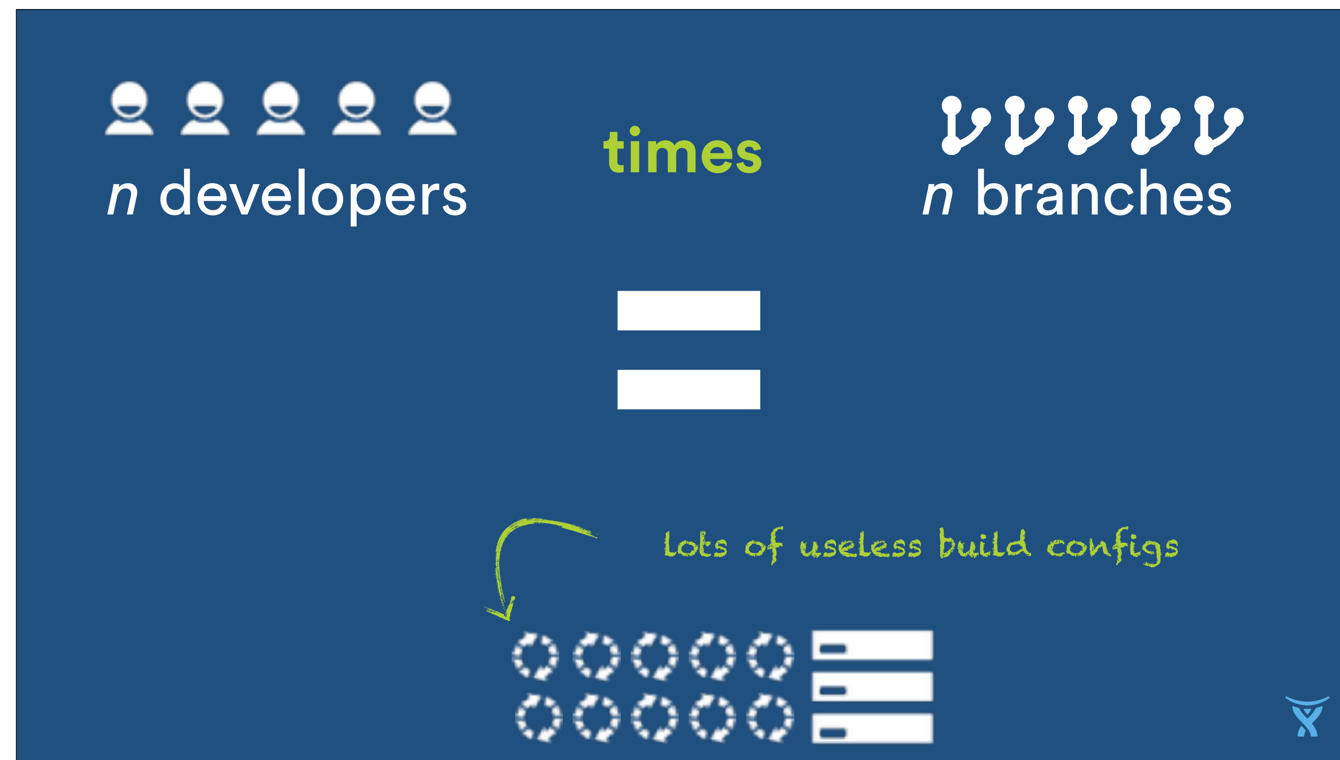
We *could* just set up another Bamboo instance with another 100 agents and transfer a bunch of the builds there. But first we backed up and asked "Is this really the best process in the first place?"

The answer was "No, not really."



What we found is that a good way to balance testing rigor with resource conservation is to make builds on the dev branches push-button. This is where most of the change activity is happening, so it’s the biggest opportunity for savings. Developers find that it fits naturally into their workflow, and they like the extra control and flexibility this gives them.

For critical branches like master and stable release branches, builds are triggered automatically. Since we use dev branches for all our work, the only commits coming into master should (in theory) be dev branches getting merged in. Plus, these are the code lines we release from and make our dev branches from, so for all of those reasons, it’s really important that we get test results from every commit on them, and get them right away.



So that solves one problem. But another problem remains.

Since feature branches live only a short time, their builds are active for only a short time.

So if we do the math again… (CLICK)

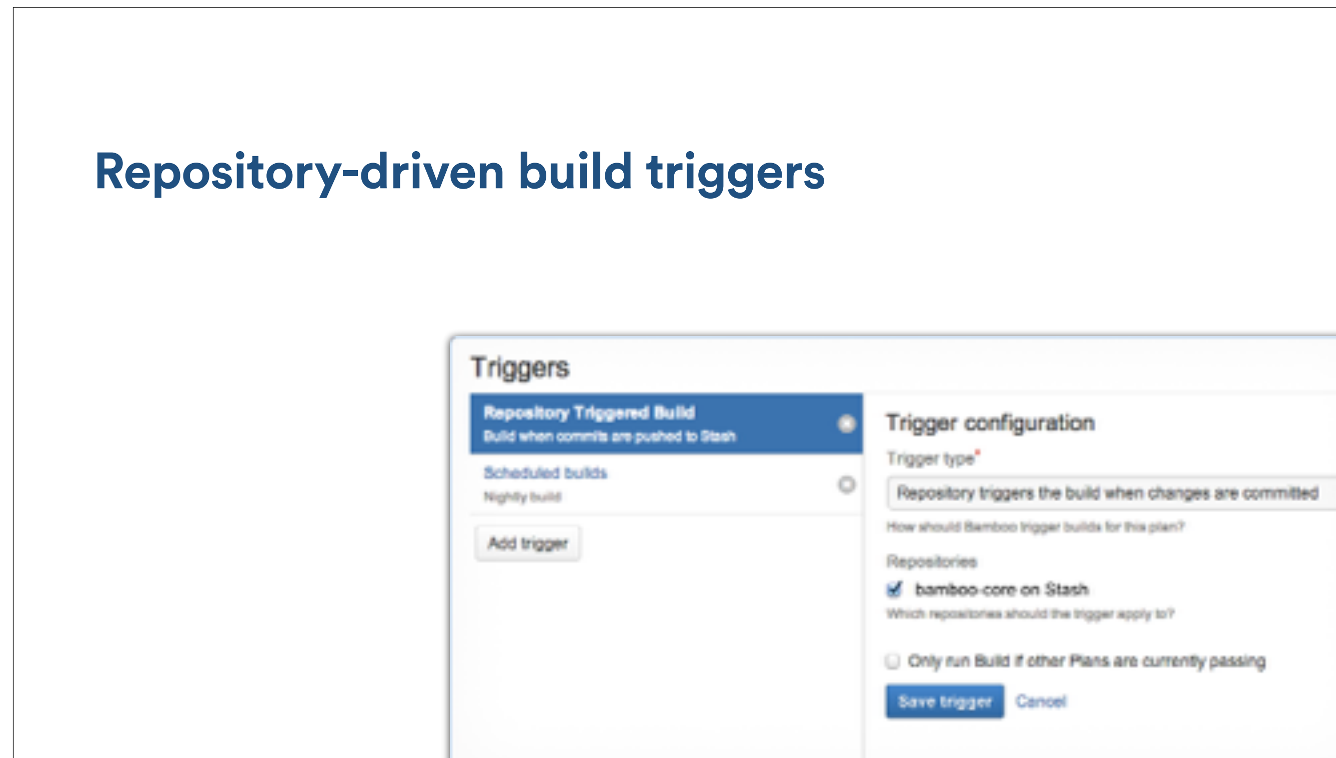
All your developers times all their short-lived branches… over the course of a few months, that's going to equal a lot of abandoned branches. (CLICK) AND… a lot of build configs for them.

So, if you go the route of building each commit on your branches, and your branch builds are polling the repo every few minutes looking for changes, it's important to disable those builds once the branch has been merged into master. Otherwise, if your CI server is polling repos on behalf of abandoned branch builds, that's going to drag down the server's performance over time.

You'll forget to disable the branch build

But this is another thing that is really easy to forget.

So we built that into Bamboo too. You can tell it to stop polling for changes on a branch after so many days of inactivity – 5 days, 15 days… whatever.



Another option is move away from polling altogether, and have the repo call out to your CI server when a change has been pushed and needs to be built. Typically, this is done by way of a hook in your repository.

And as it happens, we recently added an integration between Stash and Bamboo that makes this extra set-up unnecessary. Once Bamboo and Stash are linked on the back end these repo-driven build triggers just work right out of the box. No hooks or special configs required.

But again: you can do this with whatever tooling you're using.

The point is that you won't need to worry about disabling branch builds because they'll simply stop getting triggered when the branch is deleted or abandoned.

Now, we've covered a lot of ground already: repos, cloning, triggers… and at this point, I want to shift gears a bit.

But it’s **not pure** CI!

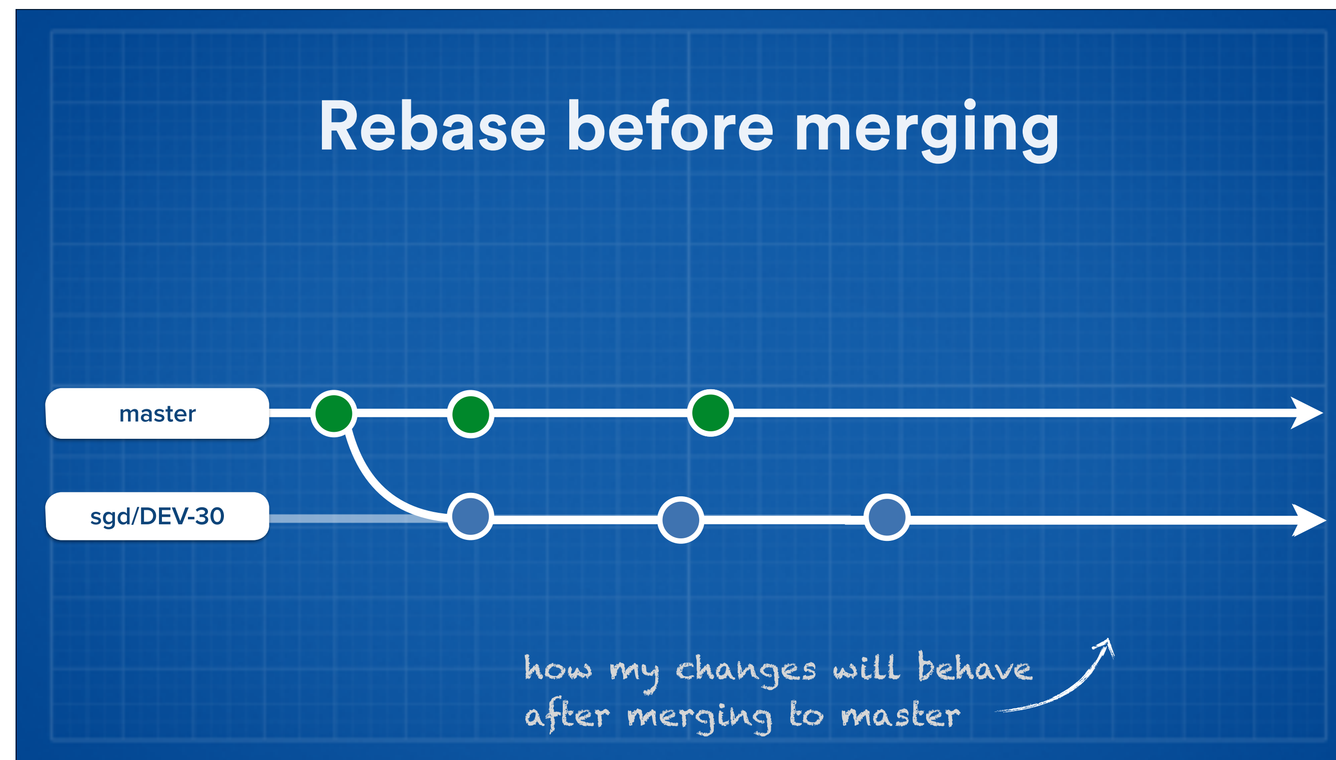
A common objection people raise when we’re talking about testing and dev branches is that it’s not “pure” CI.

And that’s true. You don’t have everyone’s changes stewing together on one code line. But that’s kind of the point. Using a single code line tends to make that code line rather unstable.

But pure CI does carry the advantage of knowing right away if your changes play nicely with my changes.

So how do we get the best of both worlds? How do we get feedback about integration without creating chaos on a critical branch?

I’m gonna talk about three different methods that you can kind of mix n’ match.



The first is to rebase your dev branch during development.

Rebase re-sets commit of origin for your branch, then the new changes you've made on the branch are re-played on top of that. (CLICK)

Running a CI build against your rebased branch provides a pretty accurate preview of how the new code will work with what's already on master. So it's a good idea to do this once or twice a day, and for sure right before you merge to master or your release branch.

Rebase is best for when you're the only person working on that branch - otherwise, it'll mess up everyone else who is working on that branch and force them to reconcile the clones on their workstations with what's now in the repo. And they will probably get very grumpy with you for making them spend time untangling all that.

Another popular option is to have your CI server merge master into your feature branch each time your build (or at least once a day).

This solves a couple of problems: it prevents your branch from drifting away from master, and it shows you how your changes will interact with the work the rest of your team is doing.

There’s one caveat, tho.

Shallow clones for the win

Remember what I said a few minutes ago about shallow clones?

Well, in order to perform merges, you need repo history so Git can look back and find the point where the two branches diverged (called the “common ancestor”).

Obviously this is incompatible with shallow clones.

So for builds that are going to employ auto-merging, cache the repo on your build agents instead of using shallow clones.

Despite that caveat, auto-merging is still worthwhile, and offers some advantages compared to rebasing against master several times a day.

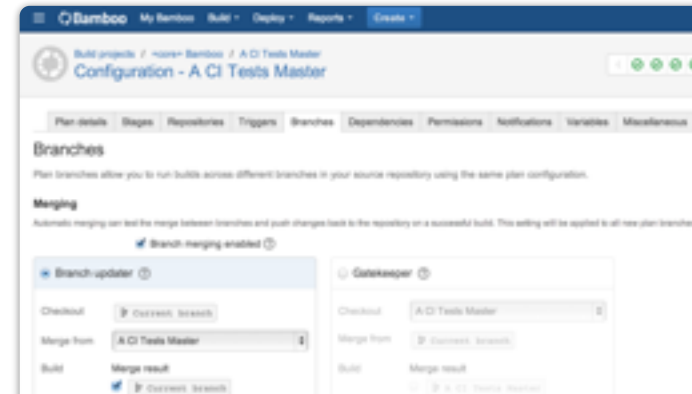
First, your history remains in tact, whereas rebase re-writes the repo’s history to make it appear as if you’d created your branch at a different point in time.

Second, you can completely automate this with your build server.

Third, when you automate this with your build server, you get to choose whether to actually push the merged code to your branch, or just use the merge as a preview.

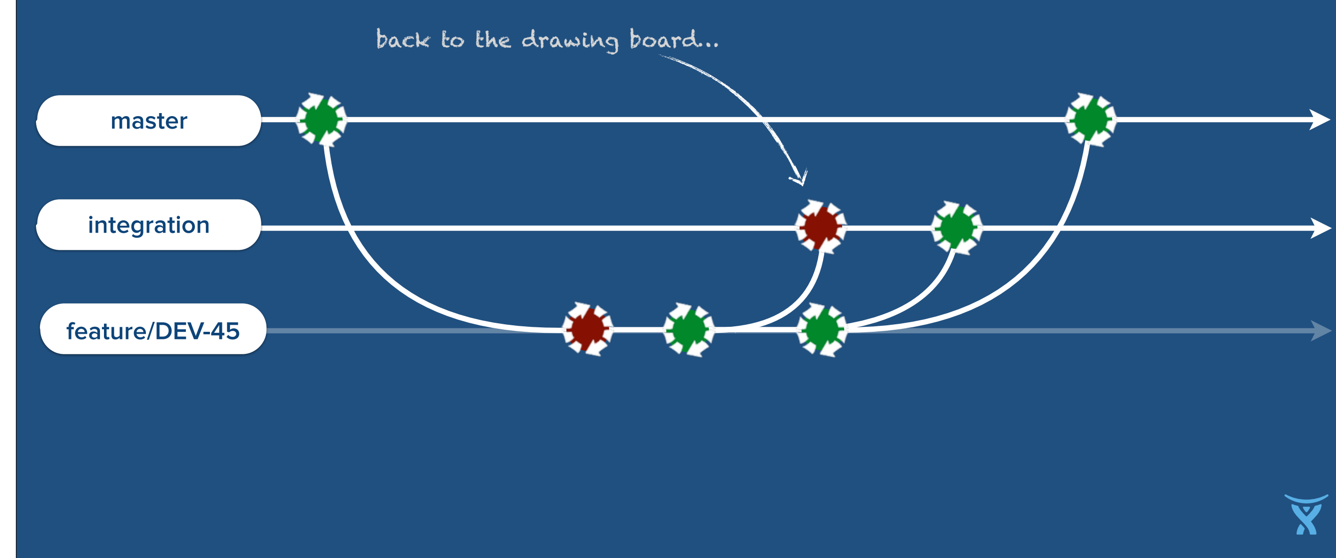
Built into

Atlassian Bamboo



And to the surprise of exactly no-one, we built that into Bamboo as well.

You enable merging for your build, select which branches you'd like to merge, and select whether you'd like to actually push that merged code back to the repo (assuming all the tests pass).



A third option that is even closer to pure CI is to include an integration branch in your team's workflow. This is similar to the "gitflow" way of doing it.

When you've completed implementation on your feature branch and you've got a clean CI run there, merge up to the integration branch. (CLICK) If your work conflicts with your teammates', go back to the feature branch and resolve the conflict.

Repeat that cycle until you get a clean CI run on both your feature branch and the shared develop branch. (CLICK)

At that point, you can merge to master.(CLICK) And it's best to merge the feature branch to master instead of merging the integration branch upstream.

Even though you just got a clean build on integration, that branch may have some half-baked features on it that aren't ready for release. If you're using feature flags, that may not matter so much. But even so: by merging the feature branch to master, stakeholders and the rest of your team will be able to see that merge in the repo history and it'll be really clear exactly what was merged in (especially if you're putting issue keys in your branch names).

Git Hooks for CI

WHY GIT?

CI-FRIENDLY REPOS

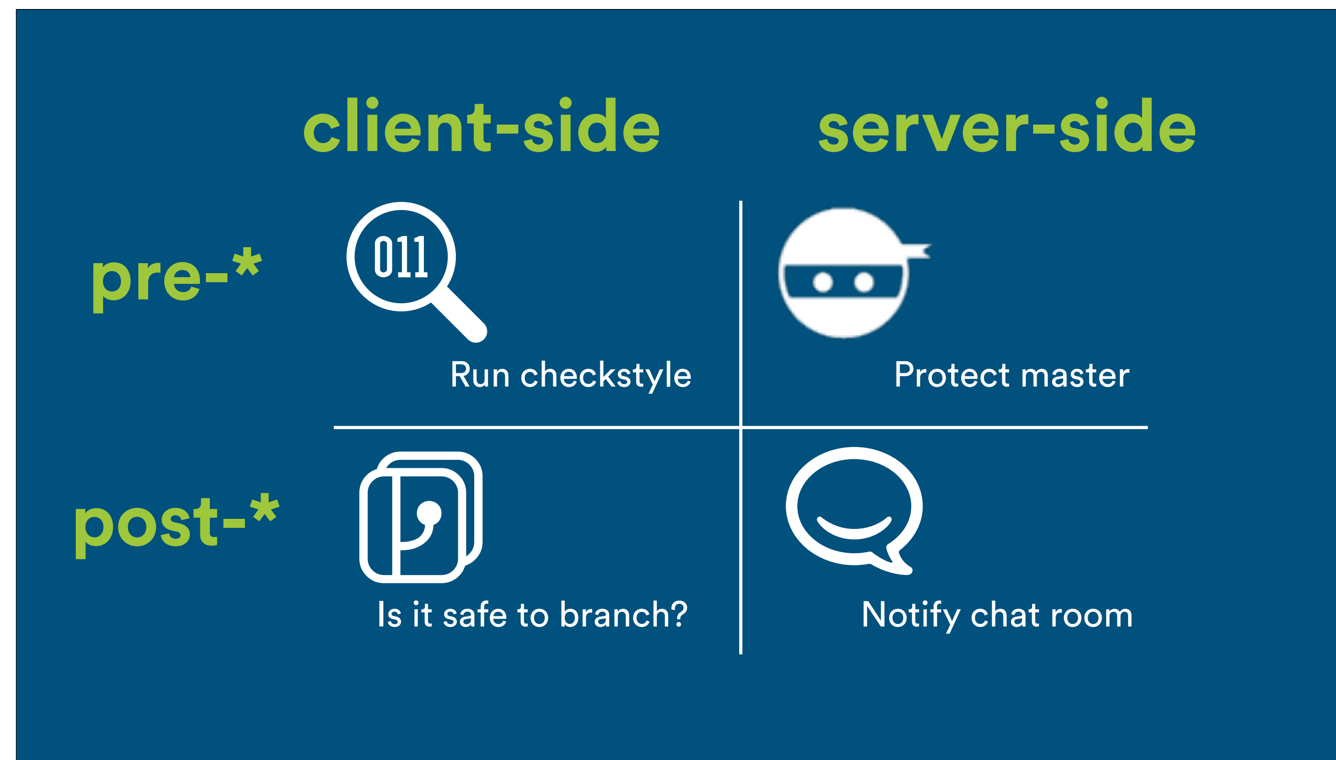
CI ON FEATURE BRANCHES

GIT HOOKS FOR CI

I mentioned hooks a couple of times already, and I want to dig into them a bit more and tell you what they are and how to use them.

.git/**hooks**

You can think of hooks as git's plugin system. They let you hook into certain git operations and customize the behavior of your repository. If you look in the .git directory of any git repository you'll see a directory named "hooks" which contains a set of example hook scripts.



There are two broad classes of hooks, client-side and server-side. Client-side hooks run on your developers machines and server-side hooks run on your git server.

You can also categorize hooks as pre- or post- hooks. Pre-hooks are invoked before certain git operations, and have the option to cancel an operation if needed. Post-hooks on the other hand run after an operation has completed, and therefore don't have the option to cancel it.

Hooks automate things like…
- checking to see if you included the key for the associated issue in your commit message
- enforcing preconditions for merging
- sending notifications to your team's chat room
- setting up your workspace after switching over to a different branch

Server-side pre-receive hooks are a really powerful compliment to CI. Pre-receive hooks can be used to prevent developers from pushing code to your server, unless the code meets certain conditions. You can think of these hooks as elite ninja guardians, protecting the master branch from bad code.

Build a fortress for master

Devs are generally conscientious enough not to merge to master when there are broken tests on their branch. But sometimes we forget to check, or if we're sharing a dev branch with other people, sometimes more changes get made since we last checked the branch build… whatever.

To give master a little extra protection, we can add a hook that looks for incoming merges to master. When it finds one, the script checks the latest build on your branch, and if there are any failing tests, the merge is rejected.

My colleague, Tim Pettersen, wrote this hook script and made it available on Bitbucket. You can grab it, customize it, and add it to your repo.

Something I've seen lots of teams struggle with is maintaining code coverage. Often times they've had to retroactively cover their code base with tests, and it's really frustrating to see that hard-earned coverage slip away as more features get added without tests shoring them up.

Tim also wrote a hook to protect master from ever-declining code coverage.

This hook also looks for incoming merges to master. It then calls out to the CI server to check current code coverage on master, and coverage on the branch. If the branch has inferior coverage, the merge is rejected.

And just to be clear, this all assumes you already have code coverage running in one of your builds. The hook doesn't magically do that – it just looks for the coverage data in your build results.

http://**bit.do/git-ci**

If you're interested in playing around with some of these hooks, checkout this link: bit.do/git-ci. It'll take you to a Bitbucket repository with some ruby hooks for enforcing check style, code coverage and green builds on your master branch.

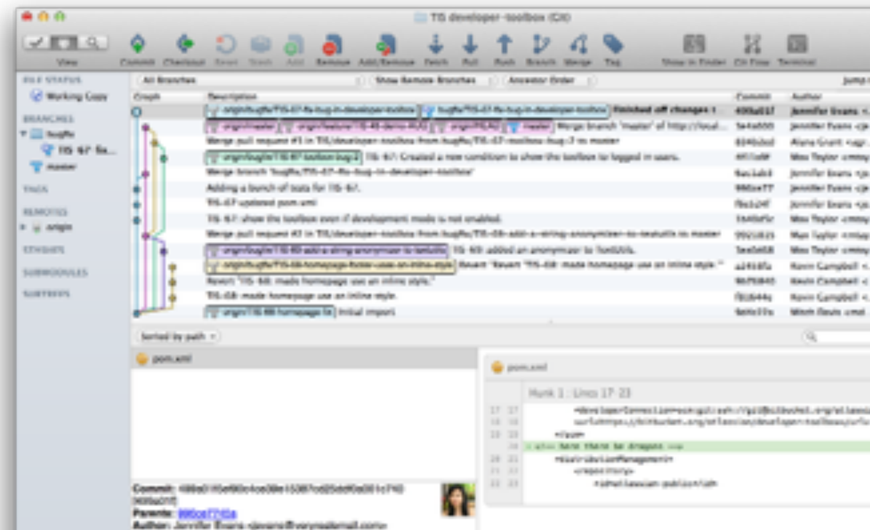
Key takeaways:

- Let feature branches do the dirty work
- Say “yes” to shallow clones and repo caching
- All your branch are belong to CI
- Share, and clean up after yourself
- Git hooks = ninja powers

So the key takeaways here are… (CLICK)
- use Git’s facility for branching and merging to keep unproven code off of master (CLICK)
- keep your repos lean, and take advantage of shallow clones & caching (CLICK)
- use automation to get CI running on all your branches and keep them updated (CLICK)
- use build resources efficiently so there’s enough to go around, and clean up after yourself when you’re done with your branch build (CLICK)
- use Git hooks to silently, invisibly protect master from human error

Free stuff!

- sourcetreeapp.com
- atlassian.com/git



I'll leave you some free stuff to check out.

We have a free app called SourceTree that's great for beginners and experts alike. It's a desktop client for Mac and Windows that provides a nice UI for commits, merges and some of the other Git operations.

If you're interested in learning more about Git, check out our tutorials site. It goes through a bunch of Git commands, different workflows, and has recordings of past Git-flavored webinars.

Q & A

And with that, let’s dig into some questions.

we're hiring!