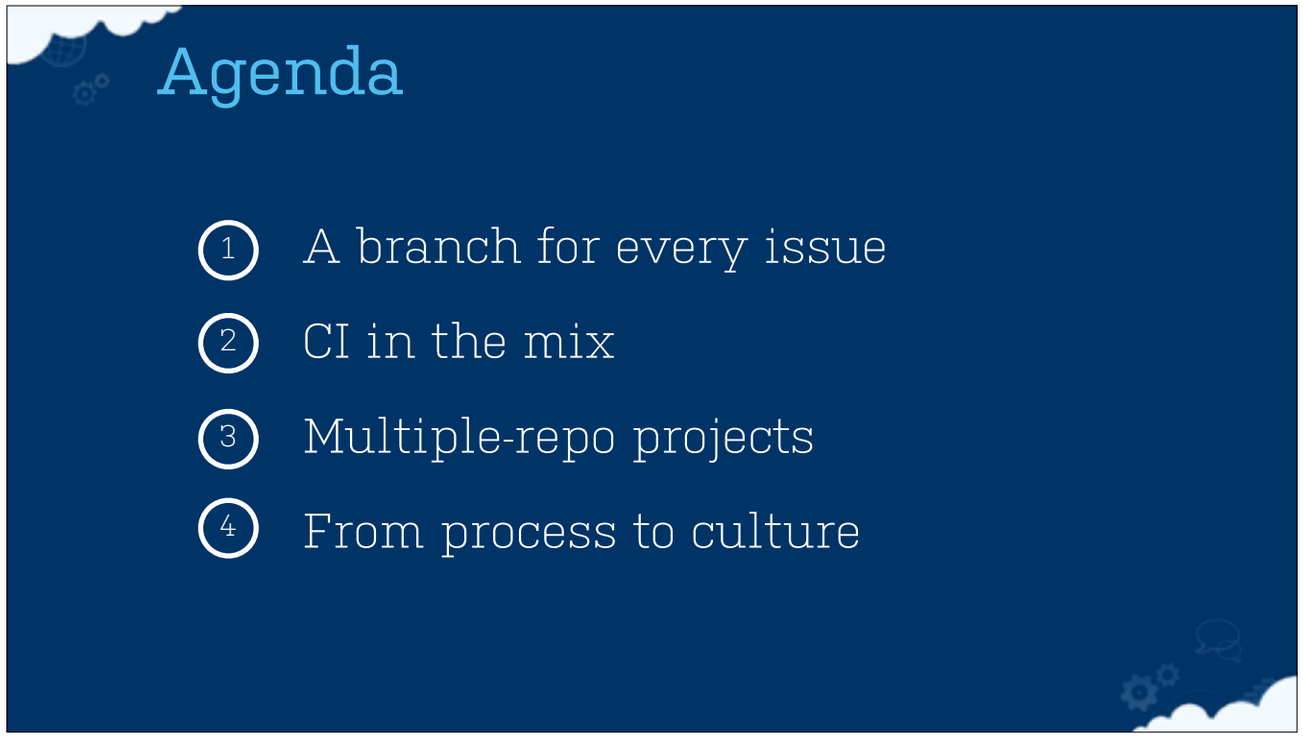

Atlassian

Hello everyone! I'll be talking about how teams at Atlassian use Git branches for continuous delivery.

My name is Sarah, and I'm on the product marketing team. Before Atlassian, I spent almost 10 years in QA, including a few years as an automated test engineer. I don't do much coding now, which is ok because I still get paid to learn, think and talk about CI, CD, automation and agile development.



I'll walk you through the branch-and-merge development workflow and talk about why we use it. Then go over some of the things we've learned about incorporating CI, working with projects that span multiple repos, and end with a few words on process and culture.

A branch for every issue

We usually think of CD in terms of killer automation, but it starts further back in the dev cycle.

Every team at Atlassian is now using Git, and we are bullish on exploiting the power of branch-and-merge workflows. In fact, we've taken to creating a dev branch for each issue we work on - be it bug, user story, or technical task. And that's helping us deliver faster than ever before.

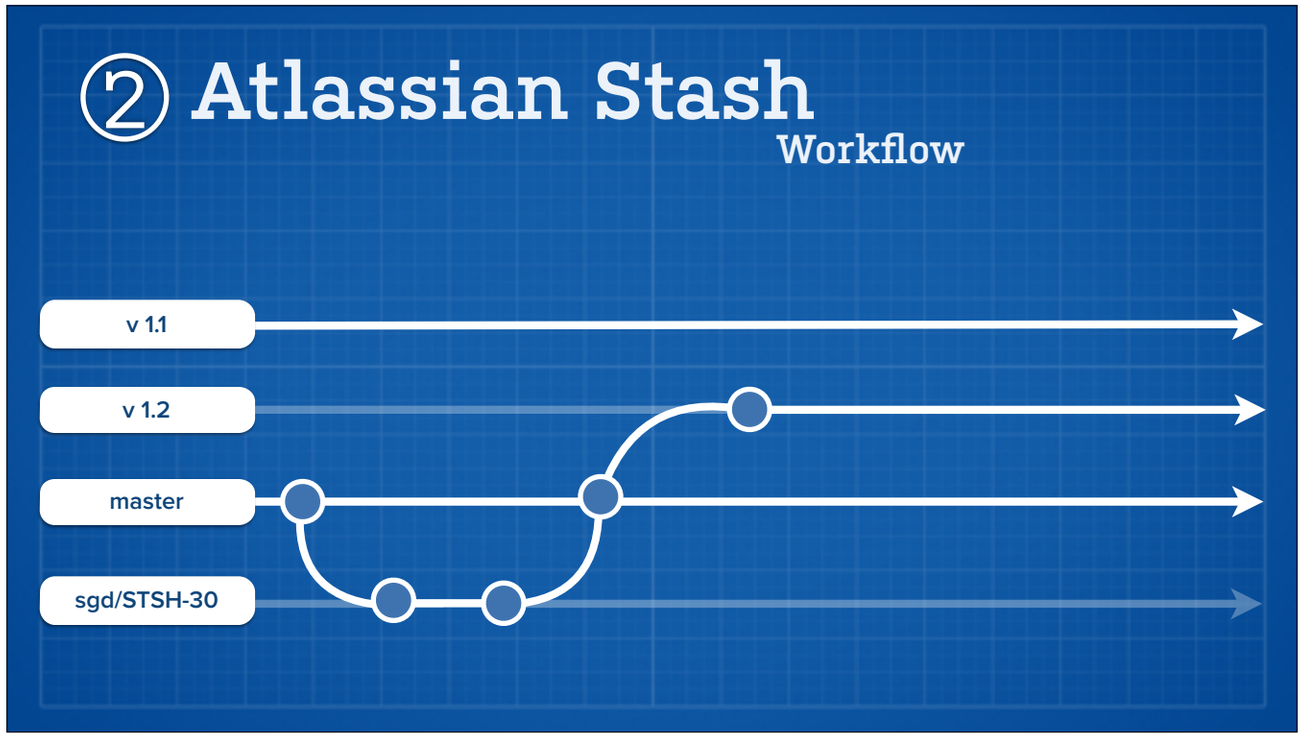
By keeping work in progress isolated on dev branches, we can keep master in a clean and releasable state. And by developing in small, modular pieces we avoid having lots of tightly coupled code that has to be released all together.

This means our SaaS teams like Marketplace and Bitbucket can release every day if they want to, and our teams that make installed products can continuously deliver to our dogfooding instances and ship to customers every 6 or 7 weeks.

If you haven’t worked with Git yet, you’ve probably heard about it. And for the next few minutes, all you need to know about Git is that branching and merging are really easy.

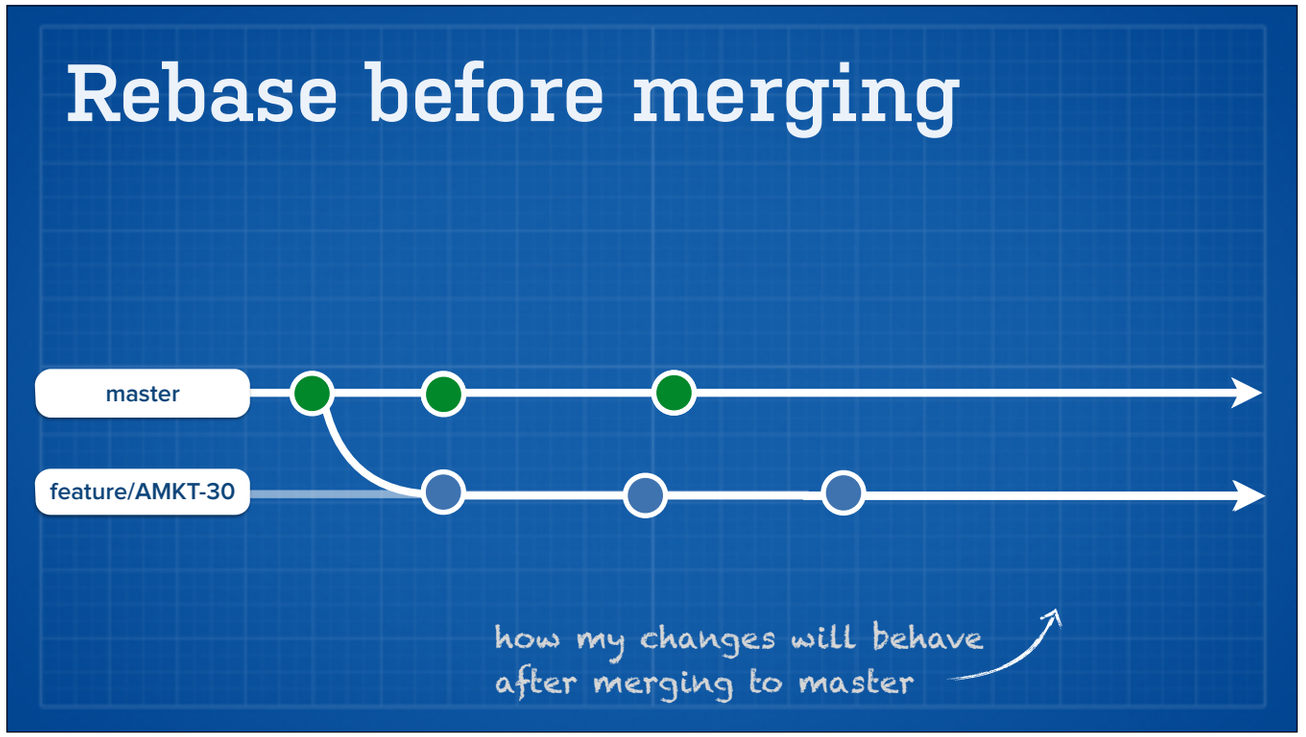
You won’t find the day-long code freezes and hellish merges and heavy administrative controls that are associated with centralized VCSs like Subversion.

(call out naming convention)


② Atlassian Stash
Workflow
v 1.1
v 1.2
master
sgd/STSH-30

```
$ git fetch origin
$ git rebase master
```

When each developer has their own branch, merge conflicts are bound to happen. We've found that doing frequent fetch & rebasing of the dev branch helps avoid surprises during the final merge to master. It also helps catch integration issues faster and lets us get those resolved down on the branch.



Almost all developers do a fetch and rebase when updating their dev branches - not a fetch merge. Rebase is like resetting the commit of origin for your branch, then the new changes on the branch are re-played on top of that. (CLICK)

This provides a more accurate preview of how the new code will work with what's already on master.

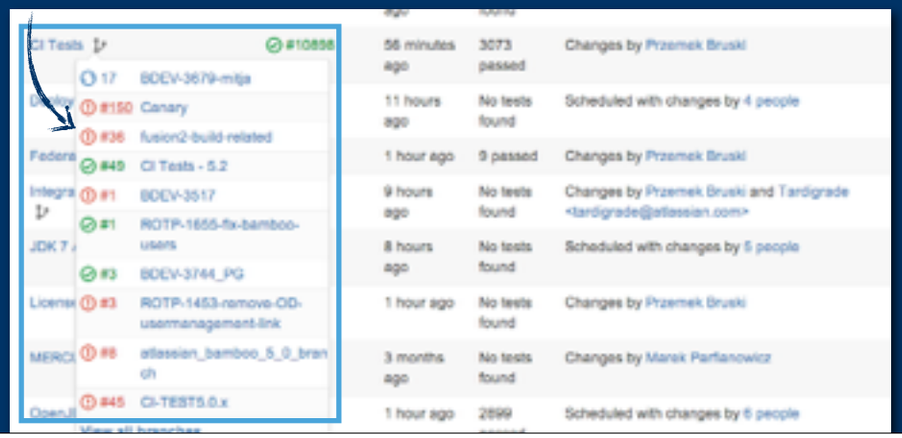
And btw, our teams prefer to use the explicit merge when dev work is merged to master or a stable release branch so the history of what was done on the branch is preserved. With fast-forward merges, you loose that.

CI in the mix

Now that we have all these branches, how does that effect continuous integration?

Running CI on dev branches

all active branches are under test

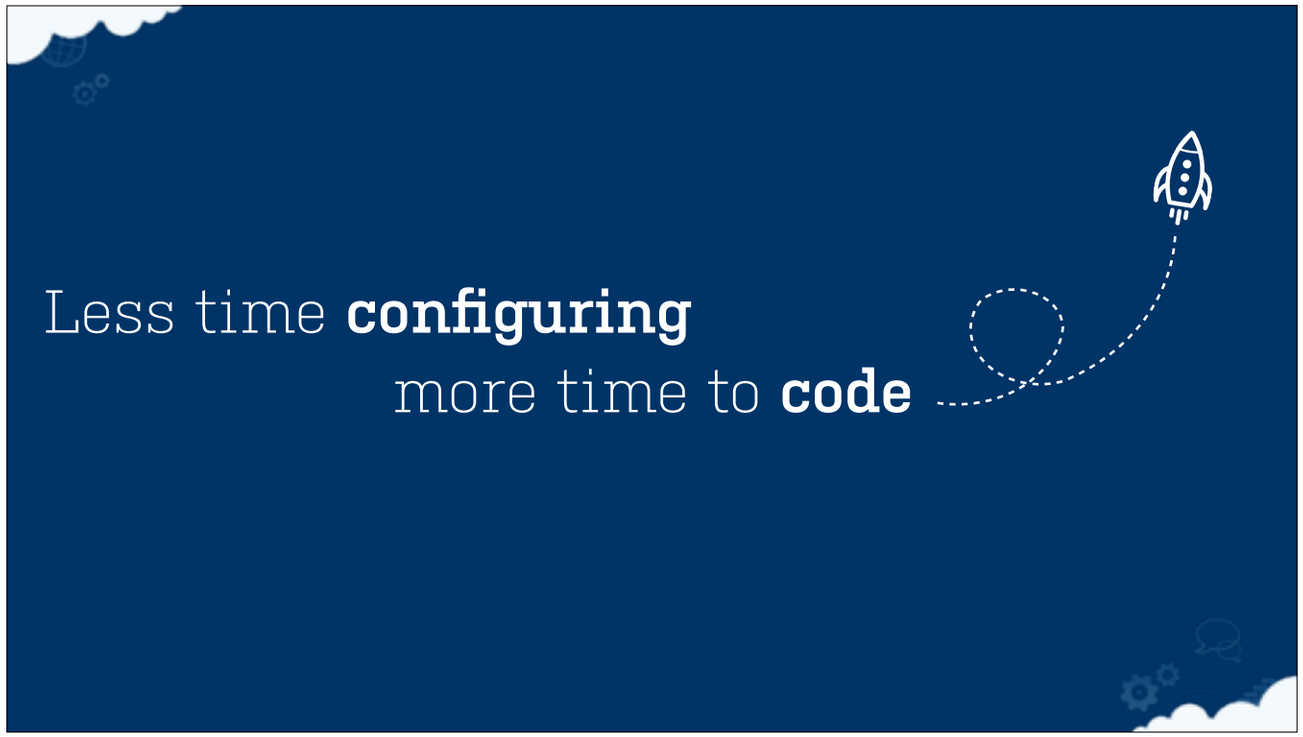


Probably goes without saying, but it’s really important to be running CI against dev branches. Otherwise you'll be hit with all kinds of surprises if you merge to master and do your first-ever CI run there - which defeats the whole purpose of this whole branching strategy.

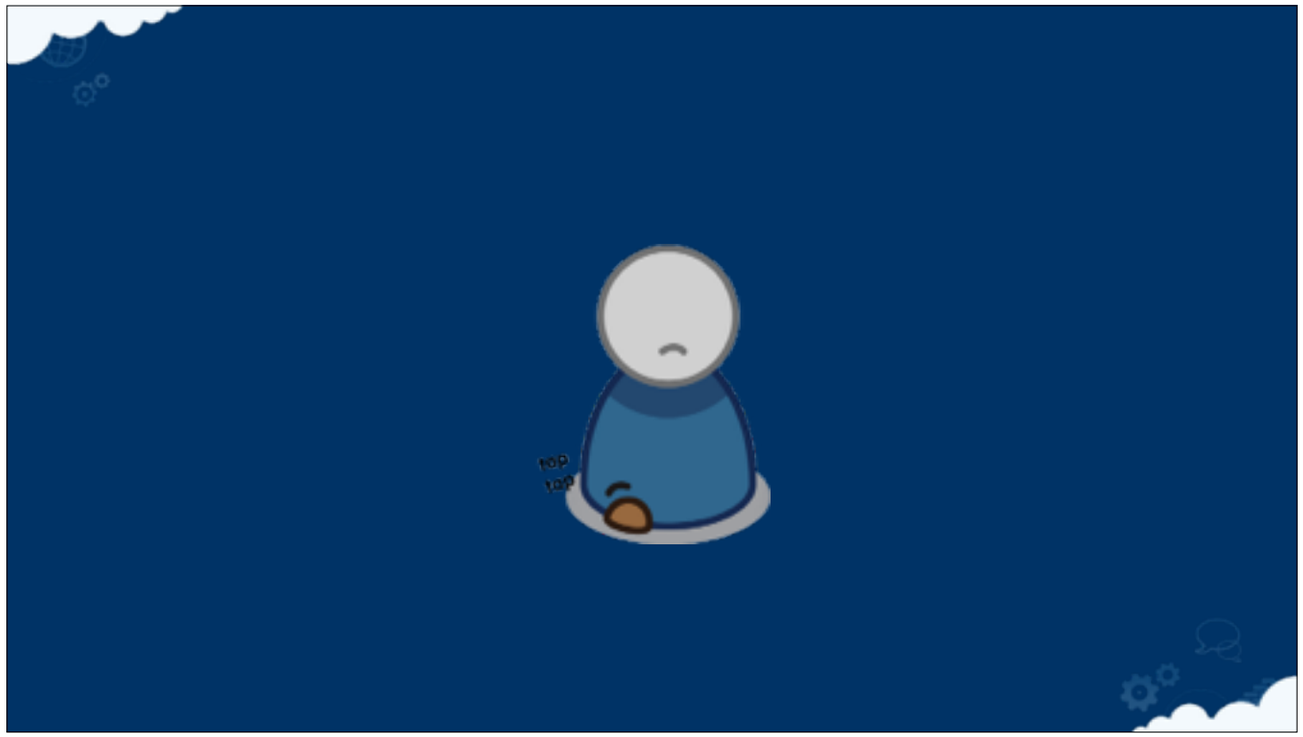
It’s pretty painful to clone your CI configs every time you start work on a new issue.

But you can automate that with a Jenkins plugin, or with a Git hook that one of our developers put up on Bitbucket.

But since we happen to make a continuous delivery server, we decided to just build that in and save ourselves (and our customers) the hassle.

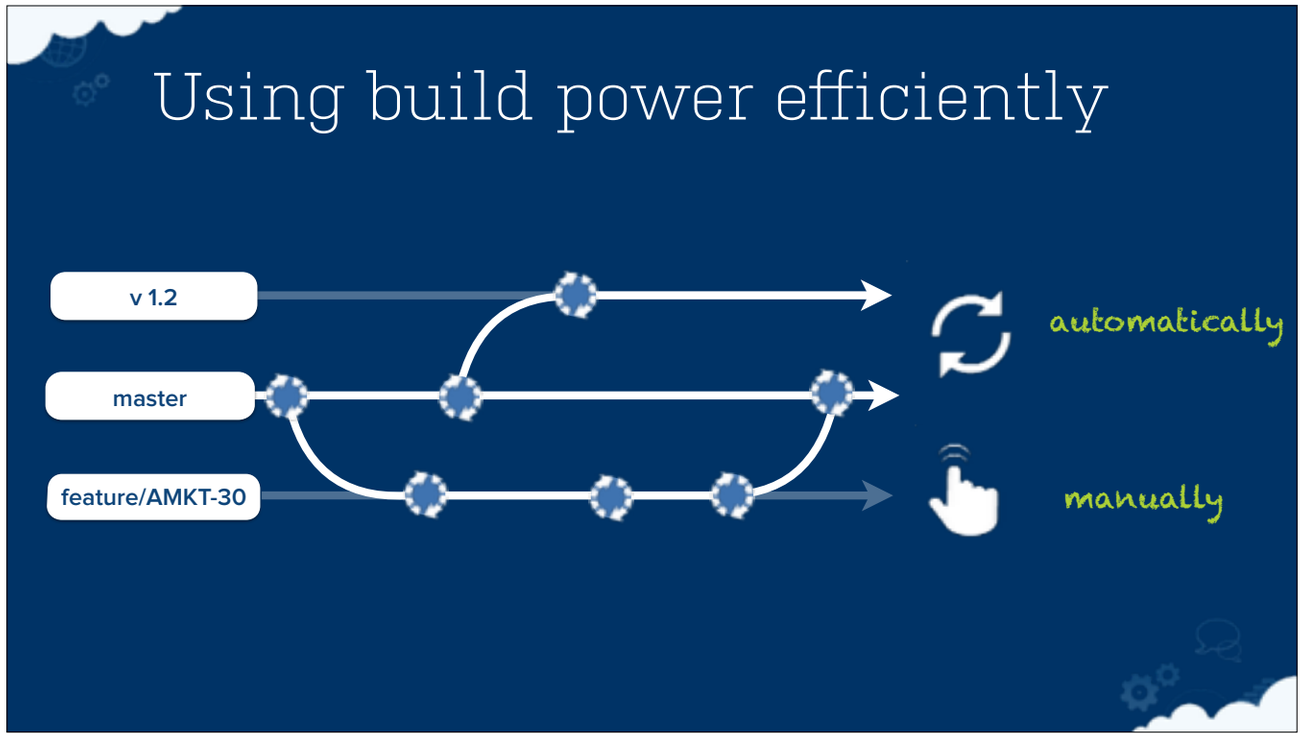


Either way, the point is to spend your time on more interesting work.



But running CI on every branch can use up a TON of resources. How do you keep your build queue from getting clogged up?

Naturally, you could increase your number of build agents (or slaves). But let's think about effeciency first.



We've found that a good way to balance testing rigor with resource conservation is to have push-button builds on development branches. This is where the most change activity is happening, and where the biggest savings can be had. Our devs find that it fits naturally into their workflow and they like the extra control/flexibility this gives them.

Builds to the most critical branches, however, are triggered automatically with each change. This includes master, and any stable release branches. Most, if not all, the changes pushed to these branches are going to be merges from dev branches, so it's important to get feedback on that quickly - every time.

Multiple-repo projects

Most of us work with multiple-repo projects, so dependency management will come into play.

We use Maven for this.

Here’s a shot of the parent pom for Stash…

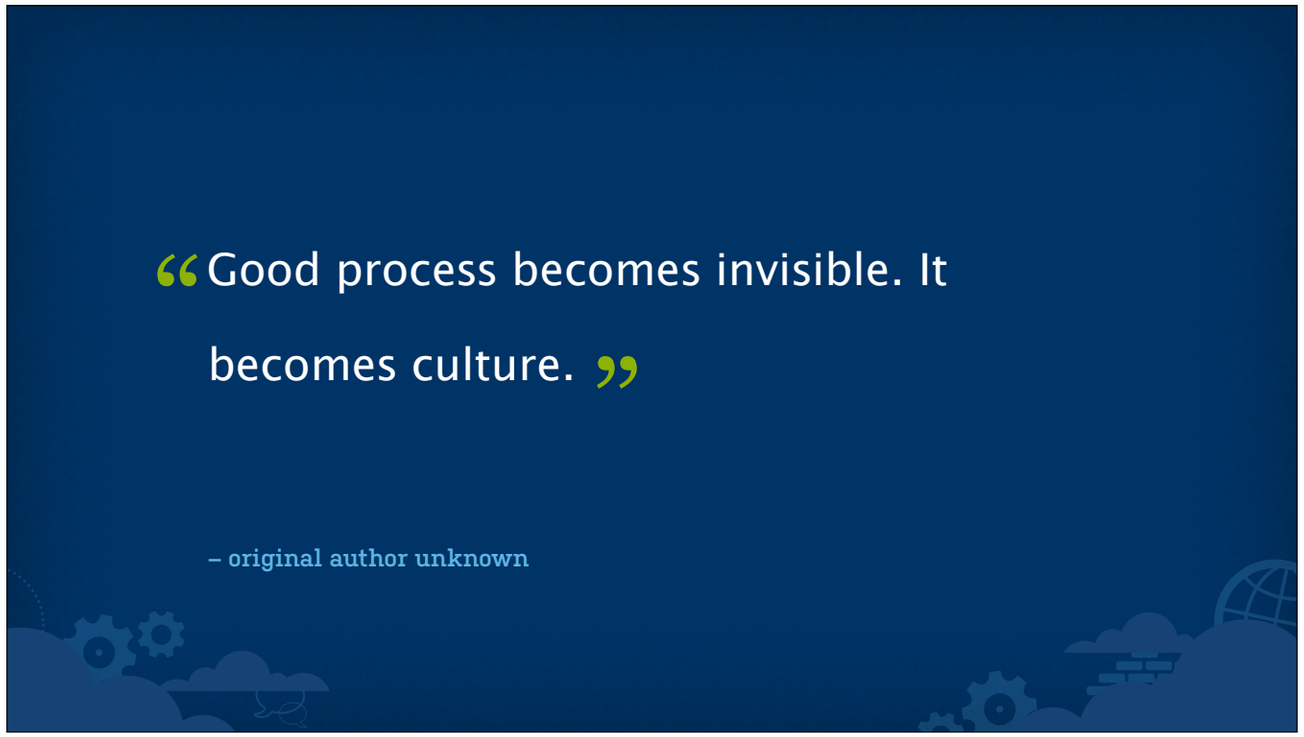
And then all the dependent modules.

So this has all the pros n’ cons of maven.

This isn’t a company standard, but as far as I know, none of the teams use Git submodules for dependency management because that would force everything into one build, and that build would take longer than we’d like. Maybe submodules are better suited for dynamic or interpreted languages.

From process to culture

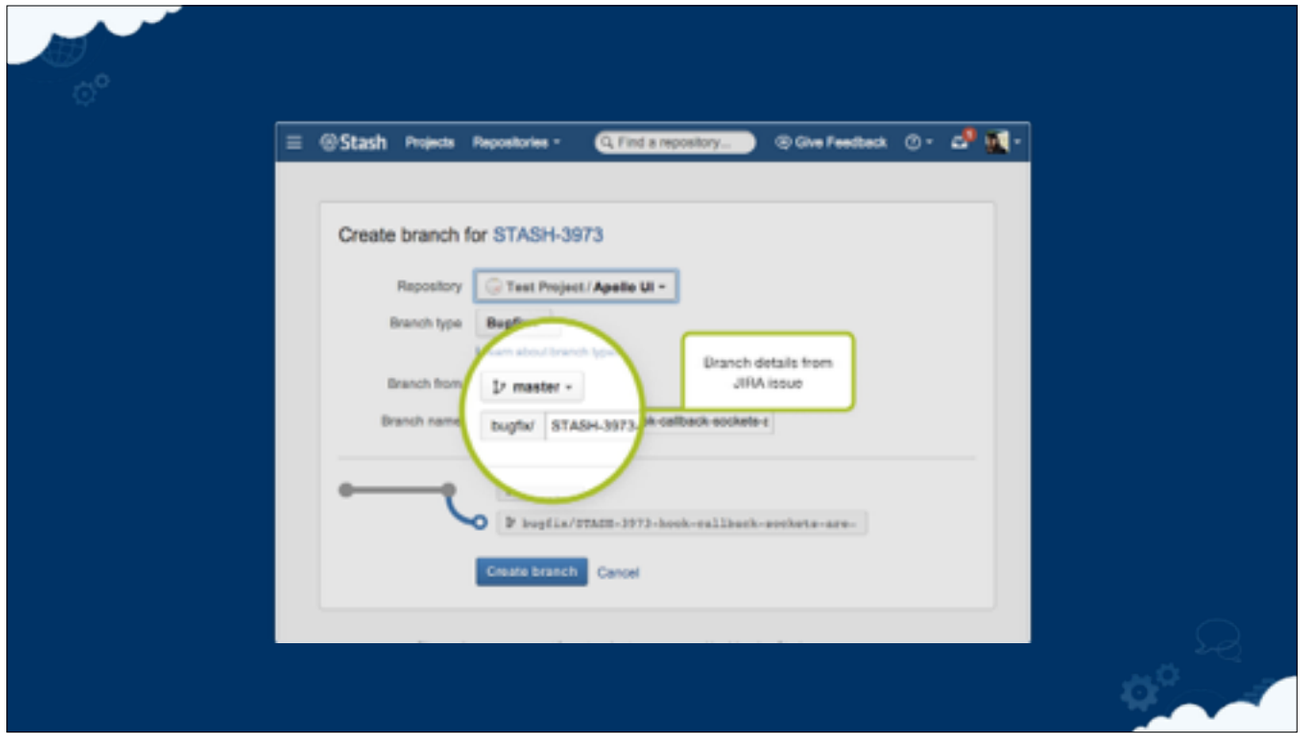
I want to end with a few words about the branch-per-issue model at a high level.



My new favorite “thing someone said about software” is that good process becomes invisible - it becomes culture.

And that perfectly describes what’s happened for us with the branch-per-issue model. For us, doing the dev work on a feature branch has become part of the definition of done.

Including the JIRA issue key in our naming convention and making use of pull requests before merging to master helped us get into the swing of it.



Then after we'd been working like this for a while, we started building features into our own tools like Stash and Bamboo and Bitbucket that make it less tedious for developers to do this.

But we've never enforced it programmatically. You could, in theory. But I don't recommend that because it's inflexible and will end up getting in your way at some point.

If you're feeling like enforcement is necessary, that's a bad smell. It probably means that the process you're about to enforce isn't a good process for your team. And no amount of tooling is going to fix that. The role of tooling is merely to take the busy-work away.

More info at...

1. http://atlassian.com/git
2. @Atlassian


we're hiring!
Atlassian