Software is too expensive to build cheaply....
Robert's Rambling Ruminations Regarding Reality...

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

I mentioned in my last post how we have started using Feature Branches with Git and Jenkins. In that, I rather casually mentioned that

> *"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 feature means that broken builds have almost no impact on team productivity. In fact – it can even be more productive to allow a broken build than to try to prevent it all the time.

How do you do it?

Let's talk the mechanics of how to do this. I'll assume that you've set up a Git repository, you've got a Jenkins server, and you've installed the Git plugin for Jenkins. If you haven't done that yet, go do it now – I'll wait. 😉

Now, go in and create a new job. In the "Source Code Management" section, you should see the Git option. Select it – you'll get a bunch of options. There's a lot here,

and you really need to understand Git to understand them, but here's the quick start version:

- Repositories – URL of Repository: enter in your Git repository's URL. Something like 'git@mygitserver:path/to/repository'.
- Branches to build – if you leave this blank, it will default to '**'. This means that every branch will get built whenever any change is committed to them. That works fine if you're using a dedicated repository, or don't have any parallel development branches (including maintenance branches). However, you can put in any glob pattern you like – for example, 'stories/**'.

Now you want to hit the 'Advanced' button. Not the *first* Advanced button, just under the 'URL of Repository' box – no, you want the *second* Advanced button, under the 'Branches to build' section. (There aren't any fancy-pants UXI designers here!) Fill in these values:

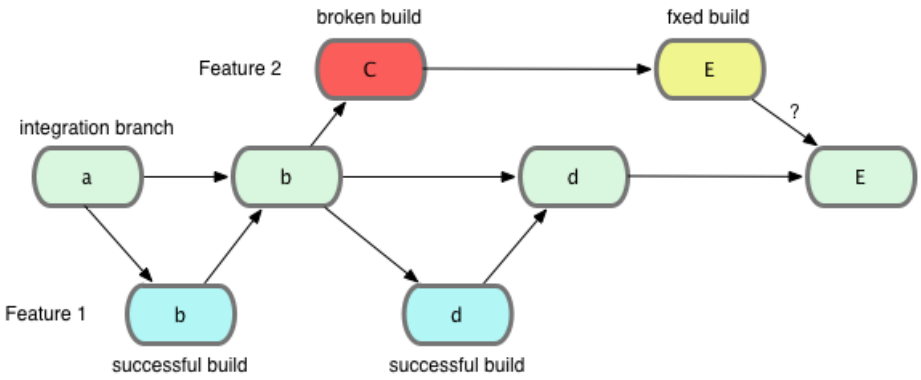
- Checkout/merge to local branch (optional): Put in the name of the branch you want to merge to. Yes, this is optional, but it means you can do things like use the Maven Release Plugin. If you don't want to do that, don't bother – this is just local to the Jenkins workspace.
- Merge options: tick the 'Merge before build' box. This will give you two more options. In the 'Branch to merge to' box, put the name of the branch in. I use 'staging' for my target branch, but you can use 'master' or whatever you want.

Okay, now go down to the 'Git Publisher' section at the bottom of the page. In this area, you want to:

- Tick the 'Git Publisher' box.
- Tick the newly revealed 'Merge Results' box.

Right! Now you're good to go. Let's talk about what will happen.

How the builds work



Here's a typical scenario: a developer creates a feature brach (Feature 1 – the bottom line) off the current HEAD of the integration branch ('a', on the middle line). She does some work, commit it and push it to the build server. The resulting build is successful, and so the change is merged back to the integration branch.

At this point, another developer creates another feature branch (Feature 2 – the top line). He does some work, commit it and push it to the build server. This build fails – the work in not merged in. Presumably, the developer will get emailed about it. With a more conventional approach to builds, this is where problems occur – people will not be able to make successful builds until something is done. However, that's not a problem with this feature!

The first developer, meanwhile, has continued on with her work, commits it and pushes it back. Jenkins sees the change and builds it – ignoring the broken branch! This then gets merged back into the integration branch, and all is good.

Finally, the second developer fixes the build. Jenkins will try to merge the changes into the integration branch – this may succeed, in which case it will build it, or it may fail with conflicts. In the latter scenario, the second developer will need to merge the integration branch into the Feature 2 branch first. However, once the branch is up-to-date (or, at least, doesn't have conflicts), Jenkins will build and fold the changes back to the integration branch.

So what can I get out out it?

Here's the big takeaway from all this: broken builds are no longer a blocker for the team – they are only a blocker for the developer (and his pair!) working on the

feature branch. As there are no impacts on the rest of the team, it becomes possible to be more lax about build failures.

> "If everything seems under control, you're just not going fast enough." – *Mario Andretti*

The purpose of the build server is to save the team time. When a broken build is a team blocker, then people want fewer broken builds – otherwise they don't get the advantage out of having a build sever, and they ignore it. But if a build *never* breaks, then the team is being too cautious – they are religiously running all the tests and other quality controls on every commit, even if there is no real chance of the checks failing. This becomes a drag by itself – even if it's a fast build.

Once you take the "broken builds block the team" out of the picture though, then a broken build becomes a tradeoff. Going to grab a drink, or take a 'bio break'? Commit and push! Time for lunch, or going home? Commit and push! Just think it's going to work, and you need to swap over to a different task? Commit and push! Let the build server find out if it worked – that's its job.

If a build is going to take, say, five minutes to run, and it will only fail one time in ten, then letting the build server do the checks will save you forty-five-minutes out of fifty. Pushing a few times a day, like everyone should? Then over a week it could save you up to two hours – that's an extra day's worth of productivity a month.

This doesn't mean that a broken build can be ignored. After all, the functionality in that branch won't be made available to anyone until the build is fixed. But it means that the broken build become non-urgent, not unimportant.

(Actually, I'd advise anyone doing this to keep an eye out for 'abandoned' branches – branches that haven't been folded back into the integration branch. This can be scripted, or made easier by tools like Gitweb or FishEye)

So that's how and why I learnt to stop being afraid of broken builds.

Related

[Making Parallel Branches meet regularly with Git and Jenkins]

In "Agile Development"

[Configuring a Multi-Branch Pipeline in Jenkins - Adventures in Learning]

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 / 20 September, 2011 / Agile Development / agile, continuous integration, git, jenkins

19 thoughts on “Git, Feature Branches, and Jenkins – or how I learned to stop worrying about broken builds”

Alexis

17 October, 2011 at 14:12

interesting workflow indeed, I’ve never paid attention to these options in Jenkins. However how does the build react when there are merge conflicts? Is it flagged as broken until the developer of the feature branch manually merges the integration branch in his branch?

don’t you end up always manually merging the integration branch before pushing, just in case the merge is not straightforward?

Stig Kleppe-Jørgensen

22 February, 2012 at 02:19

> Yes, you will have to manually merge the integration branch into your branch to get _your_ commits to build if there is a merge error. But, as he says, if another developer pushes changes and they merge ok, the build will be green again.

Joe McDonagh

22 May, 2012 at 01:35

We have the same settings AFAICT and a broken topic branch that can’t be merged causes the build to fail, and remain broken until the branch is fixed so it will merge cleanly… so I’m having trouble understanding specifically what you mean when you reference a ‘feature’ to stop this from happening… would be a god-send to have this feature.

Robert Watkins

28 May, 2012 at 22:05

Yes – there's a bug in the Git plugin where if it can't merge a branch it gets stuck. That's a bug, and it's bloody annoying.

The behaviour I describe is when the branch merges cleanly, but doesn't pass – at this point, the feature branch will not get monitored anymore. You may need to clean the workspace out between builds.

Joe McDonagh

29 May, 2012 at 10:48

Good to know that it is not my insanity.

Pingback: How to stop worrying about broken builds | Perficient Multi Shoring Blog

---

Chuck

4 April, 2014 at 02:15

I am hoping you can clarify something for us. We are using the branching model described at http://nvie.com/posts/a-successful-git-branching-model/. As described in the article, we have a "develop" branch where we integrate features that will go into our latest release. We create feature branches off of this branch where new features are developed. The majority of our feature branches are coded by one, two or three people. If we use the approach described in your article, we would turn on auto merging from our feature branches to our "develop" branch. If a feature branch fails to build, the auto merge would not take place. Thus, the only ones affected by the bad build would be the few folks working on the failed feature branch. Am I understanding you approach correctly? Would really appreciate you help clarifying this. Thanks!!

Robert Watkins

4 April, 2014 at 06:58

Yes, that's pretty much how that approach would work.

Chuck

5 April, 2014 at 00:55

Robert, would you mind expanding your answer a bit? We have a difference of opinion internally on how your article should be interpreted. When you say "pretty much", do you mean "not exactly"? If my description in the original post is slightly inaccurate, then in what way?

The way we have implemented Jenkins auto merging is as follows

We checkout our feature branches like this:
$ git checkout -b dev/myFeature/myId origin/myFeature

We push like this:
$git push origin dev/myFeature/myId

After the push command completes and we invoke "git status", we get a result that one might not expect:
$ git status
# On branch dev/myFeature/myId
# Your branch is ahead of 'origin/practice_trunk' by 1 commit.
# (use "git push" to publish your local commits)

At this point, git still thinks that we have a commit to push to origin.

This happens due to back end processing that we've implemented. When the push occurs, a temporary branch is created on the backend and jenkins attempts to

perform a build on it. Our app is quite large so a build, unfortunately, takes about 30 minutes to complete. Only after that build is successful is our pushed commit merged into origin/myFeature. We do this to avoid pushing code that breaks our build.

A downside of this is that if I am working on my feature with another person, it will take 30 minutes after I do a push for my commit to become available for him to pull. If we are in a tight loop of incorporating each others changes, these 30 minute delays can be very unproductive.

There are other other aspects of our approach that I don't prefer, but I won't go into those so as not to bore you too much.

My preference would be to implement auto merging as I suggested in my first post. Sorry to go on for so long. Can you clarify which implementation of auto merging you prefer? Thanks!!

Chuck

Robert Watkins

11 April, 2014 at 08:16

Sorry for taking so long to get back to you on this.

First – I don't use this setup anymore. It's not that I don't support it, but I've moved on from that job nearly 2 years ago, and I now work in a different environment with a single team and short-lived feature branches, instead of multiple teams with long-lived feature branches. Different environment, different solution.

The backend processing that you describe defeats the point of the setup I've described above. What I've suggested above is that you push your work to a branch

just for you – a personal branch. This is now immediately available for others to work with if they choose – at their risk.

Jenkins will see the personal branch changes (assuming the branch name matches the pattern it is monitoring), and build it. If the build passes, it can auto-merge into the mainline branch for that project – e.g. the long-lived feature branch. If the build fails, the personal branch will just sit there idle until a commit comes in to fix it.

In the example you gave above, you'd be checking out the 'myFeature' branch to your personal 'myFeature/myId' branch and doing your changes. You then push the 'myFeature/myId' branch to the server – and that push should complete immediately, so that the git status shows you have no outstanding changes. Jenkins will then spot the changes to the 'myFeature/myId' branch, build it, then auto-merge into 'myFeature'.

Hope that clears it up.

Chuck

11 April, 2014 at 00:19

Hi Robert, Do you have any input for us? Thanks!!

Chuck

11 April, 2014 at 23:22

Robert, that clears it up nicely. Thanks!!

Torben Knerr

6 August, 2014 at 10:10

Just wondering: how do you deal with Maven versions on the feature branch?

If you don't rename them (e.g. from 1.0-SNAPSHOT to 1.0-myfeature-SNAPSHOT) you will get into trouble.

Do you have an automated process that updates the version transparently when you create the feature branch and reverts that change when you merge it back?

Robert Watkins

6 August, 2014 at 10:25

At Wotif, where I used this setup, we had individual build boxes for long-lived project branches (see the previous post). We didn't publish SNAPSHOT builds anywhere, so there was no risk of cross-project contamination on the build server. Nonetheless we would change the snapshot version with a project prefix on everything that was part of the long-lived project, as you said. This was done manually, using the Versions Maven Plugin.

The approach documented here was for short-lived branches within the project branches – things that were expected to take a few days at most (sometimes a few minutes). We didn't feel it was necessary to change versions for this work, because it was such a short period – where multiple short-lived branches overlapped (which we tried to prevent, but did happen), they needed to work together anyway.

Robert Watkins

6 August, 2014 at 10:30

We did have one build server which changed versions transparently – this build server built every single project we had, against the master branch.

What this did, though, was change all of the dependencies on internally-generated artefacts to the latest published version – again, using the Maven Versions Plugin.

It would then build and see if everything still passed. This was used as an "alert status" to indicate that a change to a lower-level artefact had broken something downstream. These changes were not, however, merged back in.

Pingback: Git Flow Screencast | Rob Simpson

Brett Cashman

25 June, 2015 at 06:33

I like this workflow, but one question I had about it is how you ensure commits to feature branches are being properly audited such as by code review. The only thing I could think of is to adopt a fork-and-pull model on each feature branch, so that changes have to be pulled into the branch by a branch owner. But that seems like a lot of process for a workflow you describe as being useful in small teams with a lot of branch churn.

I'd be interested in hearing your thoughts on this.

Robert Watkins

25 June, 2015 at 07:11

For this workflow, I would not suggest doing pre-merge code reviews. As you say, the overhead would be too much. Code reviews would be done periodically over the life of the feature branch, with all code being reviewed prior to merging to the master.

For the team that I used this workflow on, I did code reviews every couple of days, and made a note of what the last reviewed commit was. We also did a lot of pair programming, which acted as an immediate review step.

I'm in a different team environment now – a smaller team, no need for parallel branches. And as such, we use a more classic `git-flow`-esque approach of feature branches, followed by review, followed by merge. We use Bamboo's automatic branch management to make sure that the branches are clean before merging.

Pingback: [git - El incremento de Proyecto de Maven Versión con Jenkins/Git]