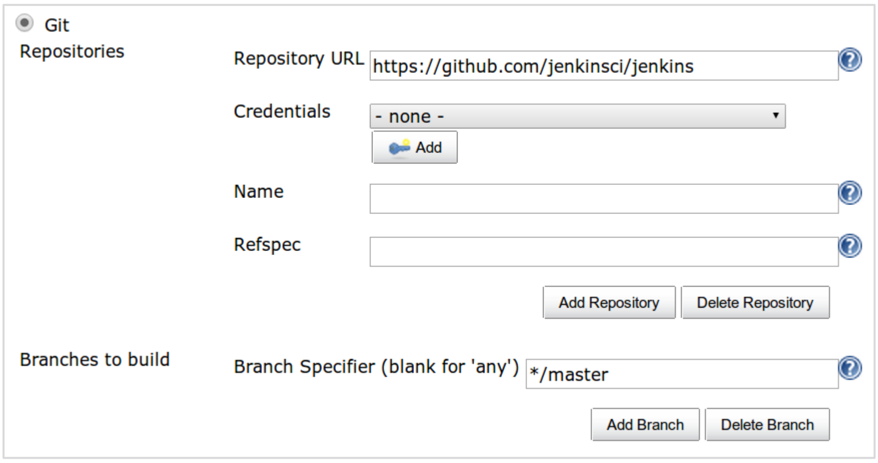
Christopher Orr
iosphere GmbH

This lightning talk is about some of the maybe less known behaviours possible with Jenkins and the Git Plugin, based on our usage of Jenkins at iosphere.

I'm Chris. I work primarily as an Android developer in Cologne for iosphere, where we build mobile applications, and where we're big fans of Jenkins.

What to build?


● Git

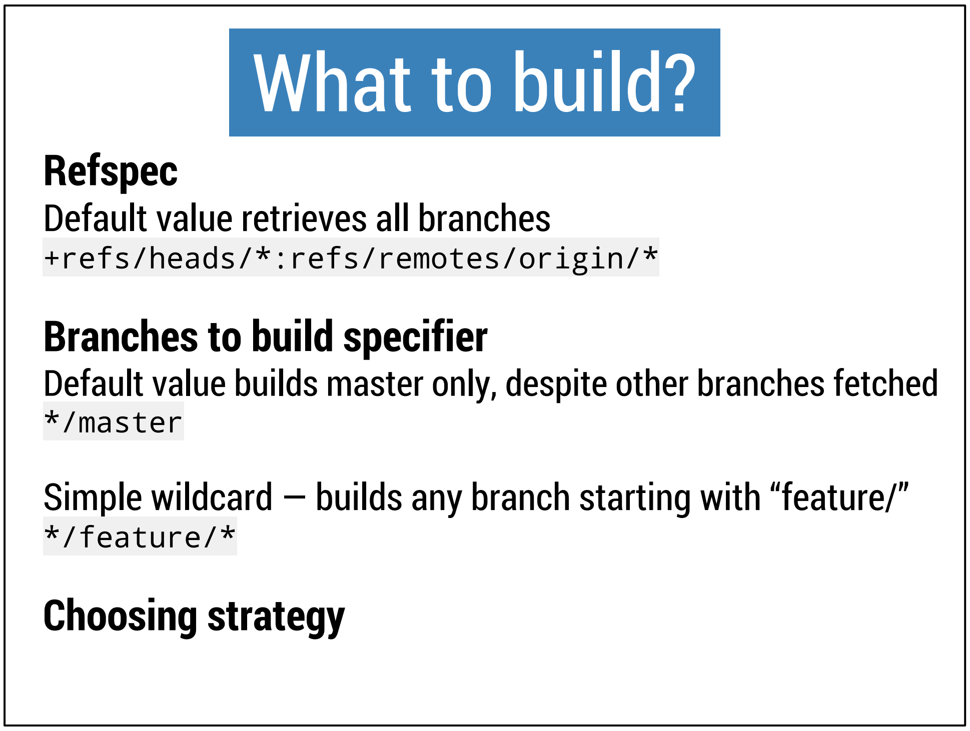
Repositories

Repository URL	https://github.com/jenkinsci/jenkins
Credentials	- none -
Name	
Refspec	

Branches to build

Branch Specifier (blank for 'any')	*/master

How does Jenkins know exactly what to build from all the various items in your git repo?



What to build?

Refspec

Default value retrieves all branches

`+refs/heads/*:refs/remotes/origin/*`

Branches to build specifier

Default value builds master only, despite other branches fetched

`*/master`

Simple wildcard — builds any branch starting with “feature/”

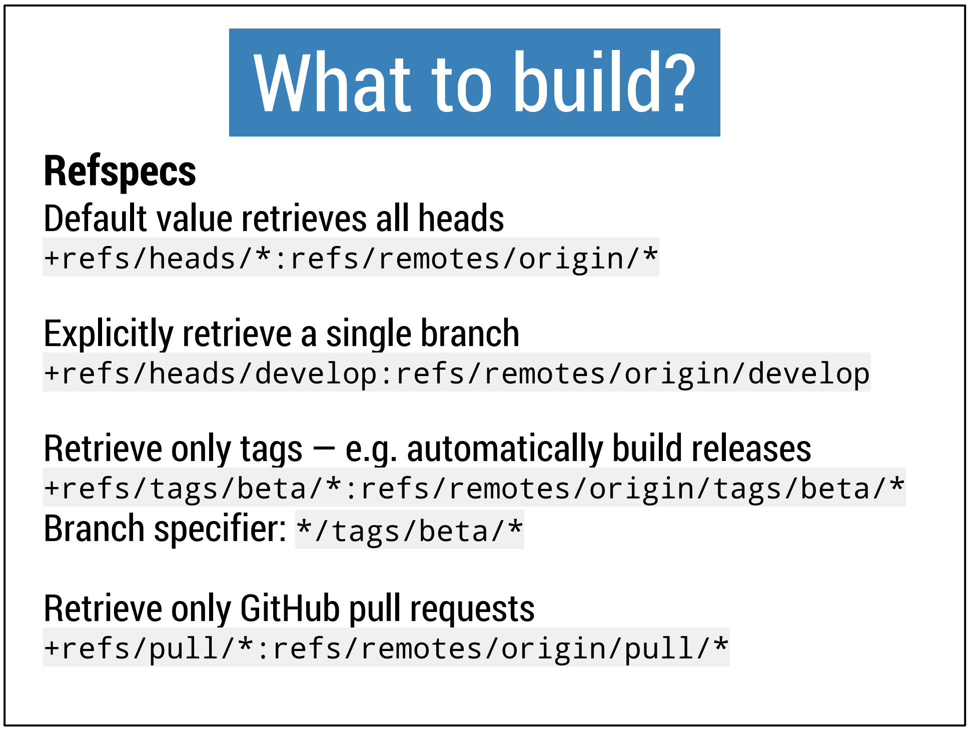
`*/feature/*`

Choosing strategy

There are three main variables used when deciding what to build, which more or less depend on one another.

The most common one is the “Branch specifier”, which lets us use simple wildcards — but not regular expressions etc.

This can be handy if you’re using the common “Git flow” way of working, where developers create branches called “feature/<something>” and you want to do a quick test run in Jenkins whenever a feature branch is pushed.



What to build?

Refspecs

Default value retrieves all heads

`+refs/heads/*:refs/remotes/origin/*`

Explicitly retrieve a single branch

`+refs/heads/develop:refs/remotes/origin/develop`

Retrieve only tags — e.g. automatically build releases

`+refs/tags/beta/*:refs/remotes/origin/tags/beta/*`

Branch specifier: `*/tags/beta/*`

Retrieve only GitHub pull requests

`+refs/pull/*:refs/remotes/origin/pull/*`

Refspecs tell git which references to fetch from the remote server (with the most common types of references being branches and tags), and what to name those references when storing them locally.

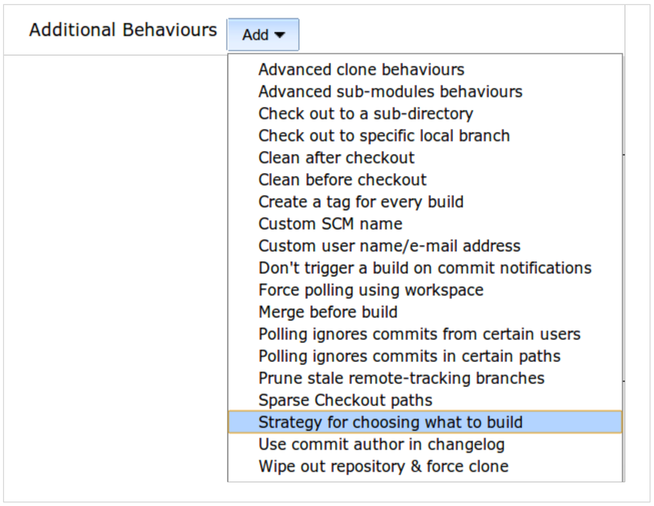
For example, rather than the default behaviour of retrieving the remote branches, we can tell Jenkins to fetch only tags by using “refs/remotes/origin/tags/*”.

At iosphere, we use this to automatically build beta versions of our software, whenever we tag any commit with “beta/<whatever>”. This is handy, as we can tag any commit like this, regardless of which branch it’s on — though typically in our git-flow setup, we will be applying the tag to the develop branch.

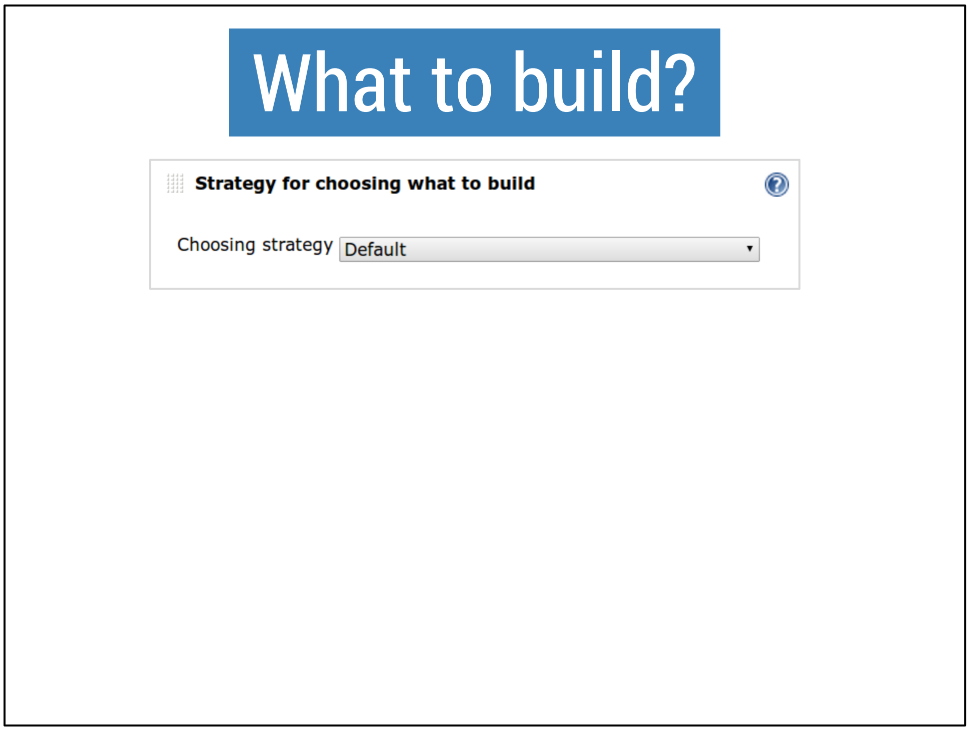
To only build beta tags, we use the refspec above, with “origin/tags” and the “beta/*” simple wildcard. Then we enter “*/tags/beta/*” as the branch specifier. This causes this Jenkins job to ignore all other branches and tags and only build when it sees a new “beta/<whatever>” tag.

Tools like GitHub or Gerrit add special reference types, e.g. GitHub pull requests are available under the “pull” hierarchy — so you could easily create a Jenkins job that monitors for new pull requests and builds them. (Though this can probably also be automated with the various GitHub plugins for Jenkins)

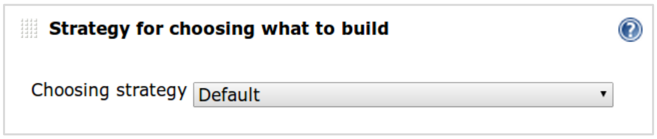
What to build?


Additional Behaviours
Add
Advanced clone behaviours
Advanced sub-modules behaviours
Check out to a sub-directory
Check out to specific local branch
Clean after checkout
Clean before checkout
Create a tag for every build
Custom SCM name
Custom user name/e-mail address
Don't trigger a build on commit notifications
Force polling using workspace
Merge before build
Polling ignores commits from certain users
Polling ignores commits in certain paths
Prune stale remote-tracking branches
Sparse Checkout paths
Strategy for choosing what to build
Use commit author in changelog
Wipe out repository & force clone

The Choosing Strategy can be found in the long list of “Additional Behaviours” that the Git plugin has.



What to build?



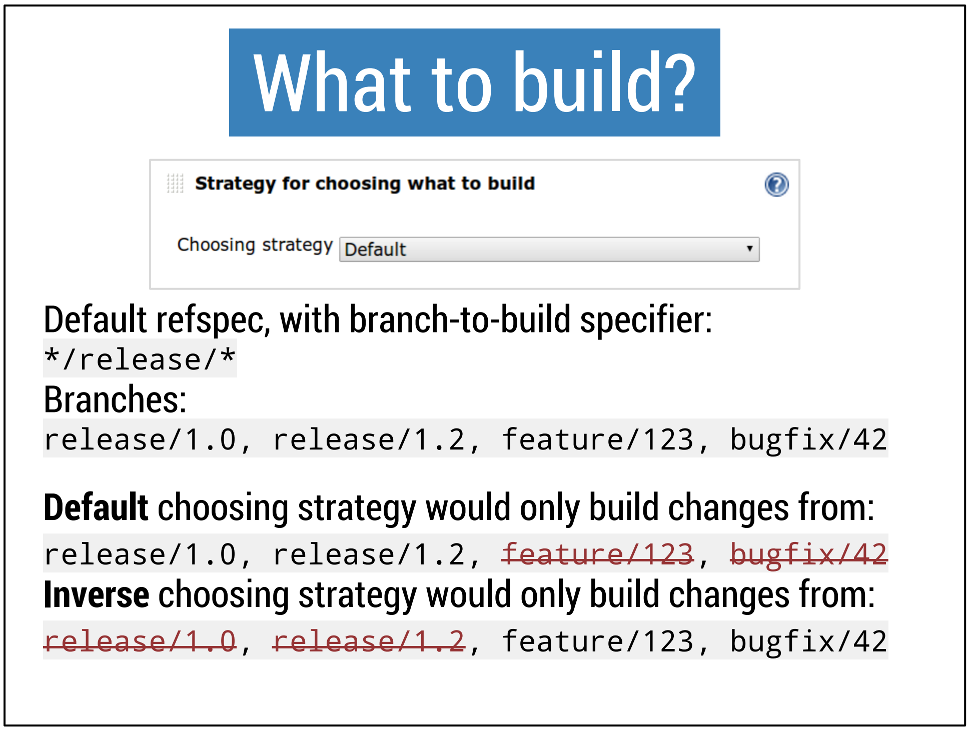
Strategy for choosing what to build

Choosing strategy: Default

The “default” choosing strategy will choose the HEAD of all branches that match the “Branch specifier”.
The branch(es) will be built in order of oldest to newest, for each branch head that has not yet been built by this job.

The other value built-in to the Git plugin is the “Inverse” choosing strategy, which basically builds everything which does **not** match the branch specifier pattern.

Choosing strategies are an extension point in Jenkins, so you can actually implement your own “BuildChooser” class in a plugin, and it will appear in this drop-down.
Plugins like Gerrit code review do this to choose build which have certain refspecs.



What to build?

Strategy for choosing what to build

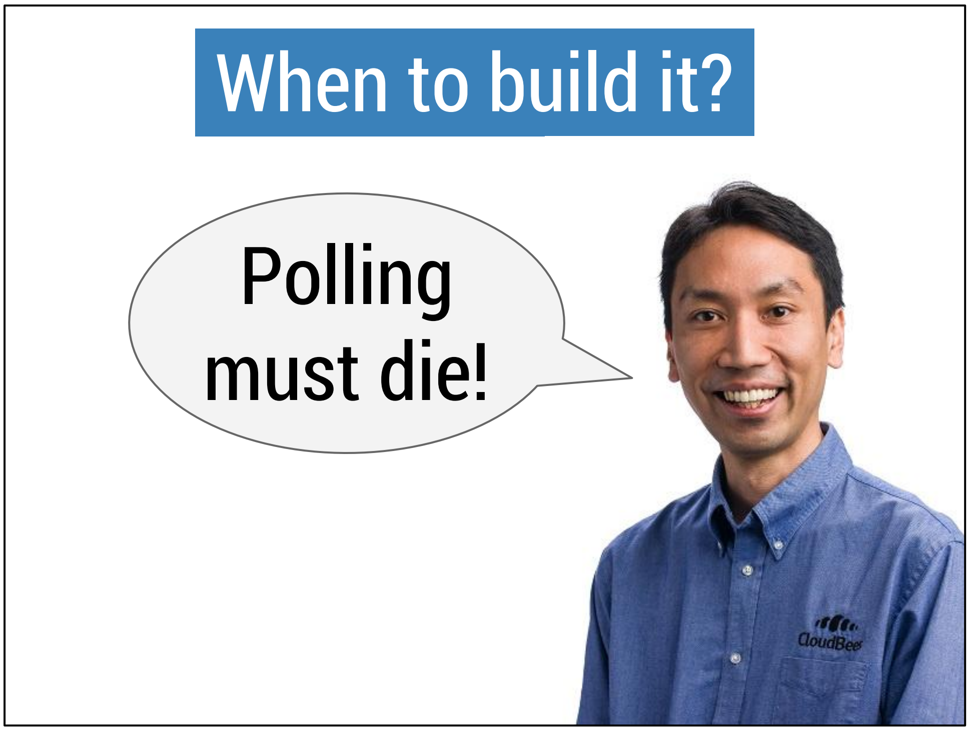
Choosing strategy: Default

Default refspec, with branch-to-build specifier:
`*/release/*`
Branches:
`release/1.0, release/1.2, feature/123, bugfix/42`

Default choosing strategy would only build changes from:
`release/1.0, release/1.2,` ~~`feature/123`~~`,` ~~`bugfix/42`~~
Inverse choosing strategy would only build changes from:
~~`release/1.0`~~`,` ~~`release/1.2`~~`, feature/123, bugfix/42`

As an example, if we configure the default refspec (which fetches all branches), and enter a branch specifier which builds all "release/*" branches:

Given we have the four branches mentioned above, the default choosing strategy would choose the two branches we expect.
The inverse strategy would do as it name suggests, and build the opposing set of branches.



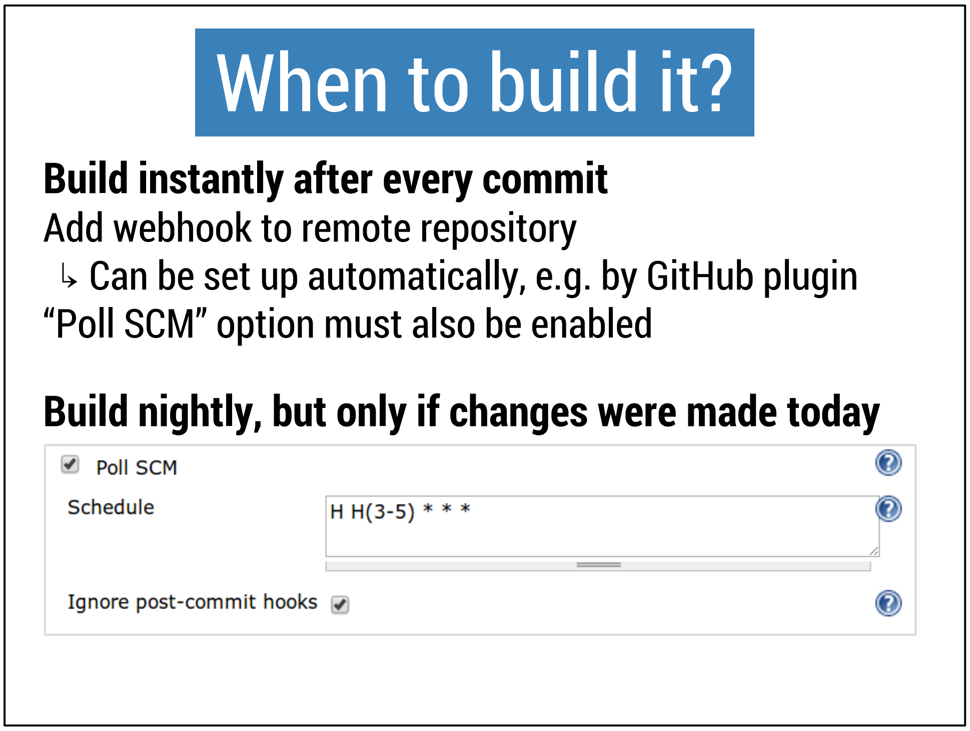
When to build it?

So we know what we want to build. But **when** should that happen?

Typically we can poll the Git repository for new changes, every 5 minutes, for example.
But this is wasteful as there are usually no changes and, as the number of Jenkins jobs increases, this can end up overloading your Git server.

Simpler, and faster is to use Git webhooks — you configure your Git server to notify Jenkins whenever a change has been made. This is more efficient, and means that jobs start as soon as possible after a commit.

Kohsuke's blog post "Polling must die", along with the Git plugin documentation covers the basics:
http://kohsuke.org/2011/12/01/polling-must-die-triggering-jenkins-builds-from-a-git-hook/



When to build it?

Build instantly after every commit

Add webhook to remote repository

↳ Can be set up automatically, e.g. by GitHub plugin

"Poll SCM" option must also be enabled

Build nightly, but only if changes were made today

☑ Poll SCM

Schedule: H H(3-5) * * *

Ignore post-commit hooks ☑

So, building instantly after every is a great idea, and can often be set up automatically for you.

One place where polling the SCM does make sense is for nightly builds, e.g. longer-running integration or performance tests.
Rather than blindly building and testing your software every night, regardless of whether the software has been changed during the day, we can use the "Poll SCM" option.
Simply enter "@daily" or "@midnight" or some other pattern, and make sure to check the "Ignore post-commit hooks" option.

This way, Jenkins will check Git once per night for changes, and only if there were any changes, a build will be triggered. The build will not be triggered by any webhooks for each commit.

Note that you should use the "Ignore post-commit hooks" option, which is SCM-agnostic, rather than the deprecated "Don't trigger a build on commit notifications" option in the Git plugin config.

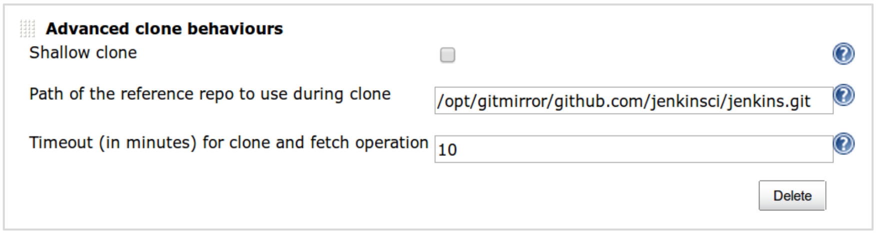
How it gets cloned

Use credentials

Avoids having to copy private keys to every slave

Automatically accepts SSH host key

Advanced clone behaviours

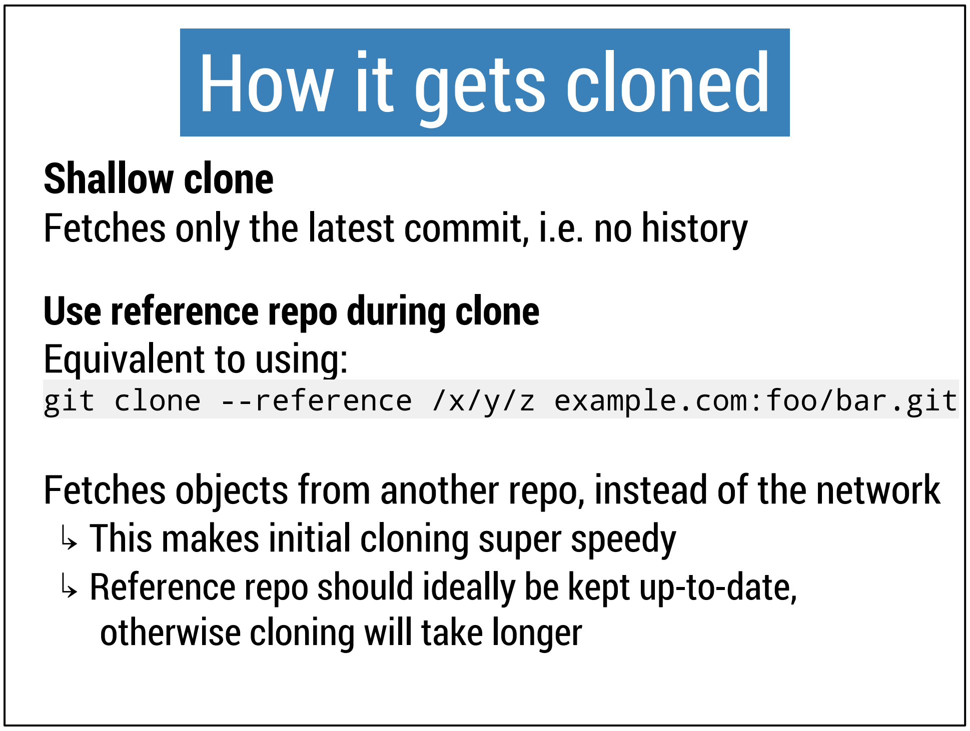


Advanced clone behaviours

Shallow clone	☐
Path of the reference repo to use during clone	/opt/gitmirror/github.com/jenkinsci/jenkins.git
Timeout (in minutes) for clone and fetch operation	10

There are also advanced options for how your code gets cloned by Jenkins.

Under the “Additional behaviours” drop-down, we can choose “Advanced clone behaviours”...



How it gets cloned

Shallow clone
Fetches only the latest commit, i.e. no history

Use reference repo during clone
Equivalent to using:

```
git clone --reference /x/y/z example.com:foo/bar.git
```

Fetches objects from another repo, instead of the network
↳ This makes initial cloning super speedy
↳ Reference repo should ideally be kept up-to-date, otherwise cloning will take longer

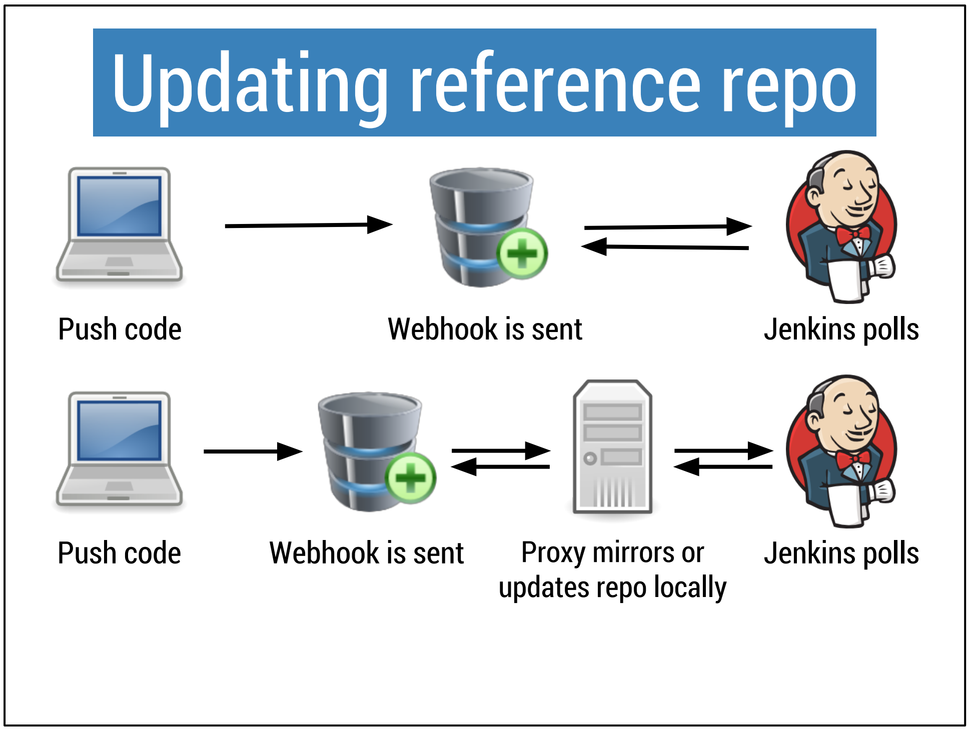
The first option is "Shallow clone", which can be helpful if you know you only want to build the latest commit on a branch — this saves some time and disk space.

The reference repo option can also help to save a lot of time and disk space — if you have many Jenkins jobs using the same repository, they can share git objects and avoid pulling hundreds of megabytes from the network for every job.

In this case "/x/y/z" — the "Path of the reference repo" refers to a filesystem path available to Jenkins containing a clone or mirror of the Git repository in question.
If this path is available during the build, Jenkins will use the "git clone --reference" behaviour and if the path is not available, Git will fall back to the default behaviour of pulling the information from the remote repository, rather than the local disk.

When this is set up, it saves a lot of time when cloning repositories.
This makes creating new jobs, setting up new slaves, or even wiping the workspace much much faster.

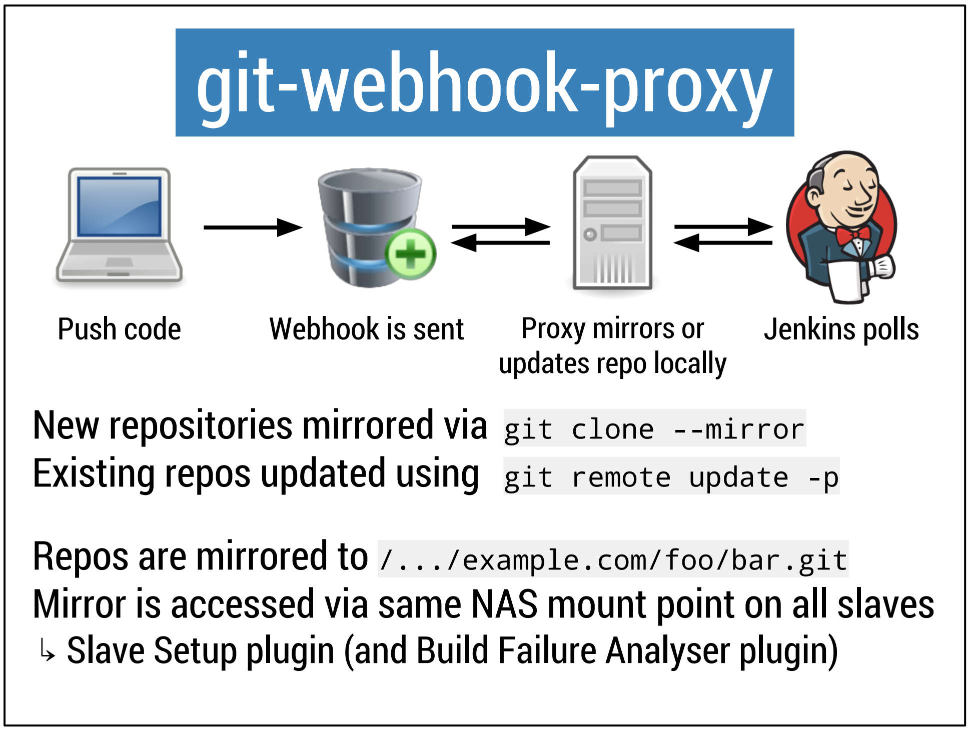
To get the maximum benefit from this feature, the reference repository should be kept up-to-date with the remote repository. If it isn't, this is no problem — Git will retrieve the majority from local disk, and the missing parts from the network — but the more up-to-date your repository is, the faster your clones will be.



Above is the normal “push code, Jenkins gets notified, Jenkins fetches code and builds” workflow.

Below is a modified version where, instead of sending git webhook notifications directly to Jenkins, we first send them to a special proxy, which examines the webhook, mirrors the repository involved locally, and then forwards the original webhook notification to Jenkins.

For the past few months at iosphere, we’ve been using this second option to automatically keep a mirror of each of git repository up-to-date.
This lets us use the “reference repo” option in all of our Jenkins jobs, and cleaning the workspace or bringing new slaves online is much faster than before.



git-webhook-proxy

Push code
Webhook is sent
Proxy mirrors or updates repo locally
Jenkins polls

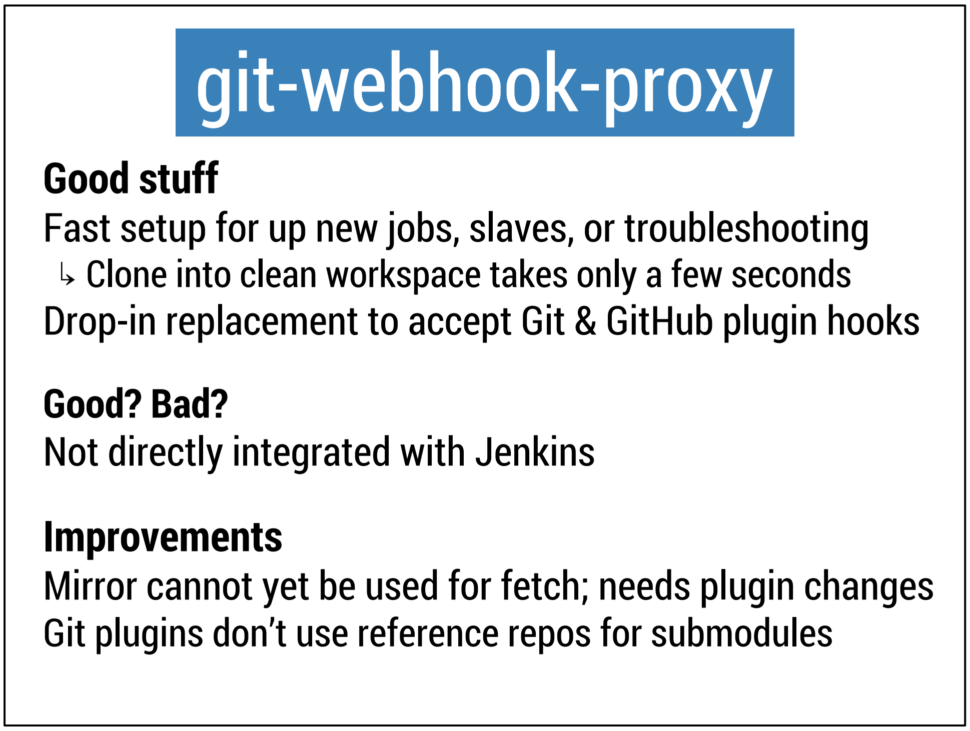
New repositories mirrored via `git clone --mirror`
Existing repos updated using `git remote update -p`

Repos are mirrored to `/.../example.com/foo/bar.git`
Mirror is accessed via same NAS mount point on all slaves
↳ Slave Setup plugin (and Build Failure Analyser plugin)

The proxy involved is a tool we built, called git-webhook-proxy.
It listens on the network for Jenkins Git or GitHub plugin webhook calls, and for each webhook, it either mirrors the repository, or updates the existing mirror.

The tool creates the mirror on a local disk, in a consistent directory layout: <host>/<path>.git — regardless of the original Git URI format ("https://example.com/foo/bar", "git@example.com/foo/bar.git" etc.).

In our case, this mirror directory is held on network-attached storage, and is made available at the same "/mnt/git/mirror" mount point on all Jenkins slaves — which is achieved by using the Slave Setup plugin.
So there is one mirror on our local network for all git repositories, and all Jenkins slaves can take advantage of it.



git-webhook-proxy

Good stuff

Fast setup for up new jobs, slaves, or troubleshooting

↳ Clone into clean workspace takes only a few seconds

Drop-in replacement to accept Git & GitHub plugin hooks

Good? Bad?

Not directly integrated with Jenkins

Improvements

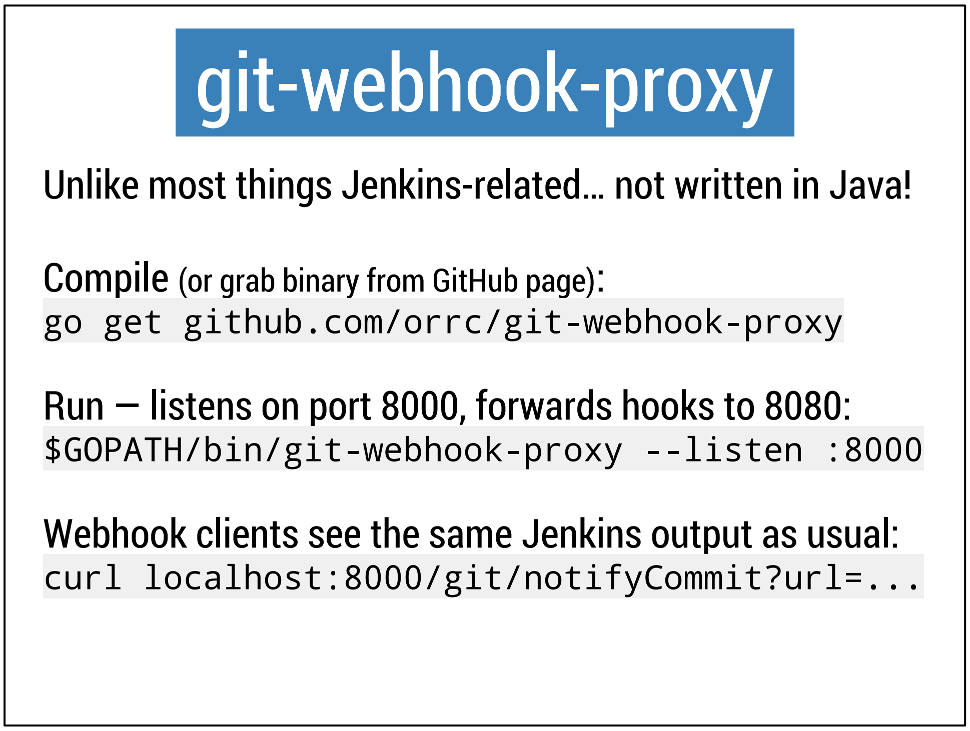
Mirror cannot yet be used for fetch; needs plugin changes

Git plugins don't use reference repos for submodules

This has been working well for us for a while now, and it makes setting up slaves (e.g. dynamically provisioned virtualised slaves) very fast, and makes fixing broken builds (e.g. by simply clearing the workspace) a lot less frustrating.
The git-webhook-proxy web server listens for the same URLs that Jenkins listens for, so it's a drop-in replacement — you can point your webhooks to git-webhook-proxy rather than Jenkins, and it will just work.

However, the reference repo behaviour doesn't currently work for submodules in the Git plugin, and the speedups only work for initial git cloning, rather than subsequent git fetches — though this could be implemented in a future Git plugin update.

So there are some positives and negatives. Some improvements could come from having this tool integrated directly with Jenkins as a plugin.
But at the same time, the software was quick to write as a simple Go server, rather than a more complicated Java plugin. Plus it can be deployed without having to be integrated with Jenkins; our git-webhook-proxy instance runs completely separate from our Jenkins master server.



git-webhook-proxy

Unlike most things Jenkins-related… not written in Java!

Compile (or grab binary from GitHub page):

```
go get github.com/orrc/git-webhook-proxy
```

Run — listens on port 8000, forwards hooks to 8080:

```
$GOPATH/bin/git-webhook-proxy --listen :8000
```

Webhook clients see the same Jenkins output as usual:

```
curl localhost:8000/git/notifyCommit?url=...
```

If you would like to try this out, the project is open source and lives on GitHub.
if you have the Go programming language tools installed, you can get this running in two steps.

The example here will start git-webhook-proxy listening on port 8000, forwarding incoming webhooks to localhost:8080 — which is the default option, but of course can be configured on the command line.

If you fetch the URL shown, with a valid Git repo URI, you'll see on the git-webhook-proxy command line that the repository is being cloned and, once complete, the curl command will show the HTTP response from Jenkins.

The end

Questions? Feedback?

chris@iosphere.de

chris.orr.me.uk/+
github.com/orrc/git-webhook-proxy

Thanks for reading / listening.

If you have any feedback, feel free to let me know!