How to build your own Git server

09 Aug 2016 | Seth Kenlon (Red Hat) (/users/seth) | 425 | 13 comments

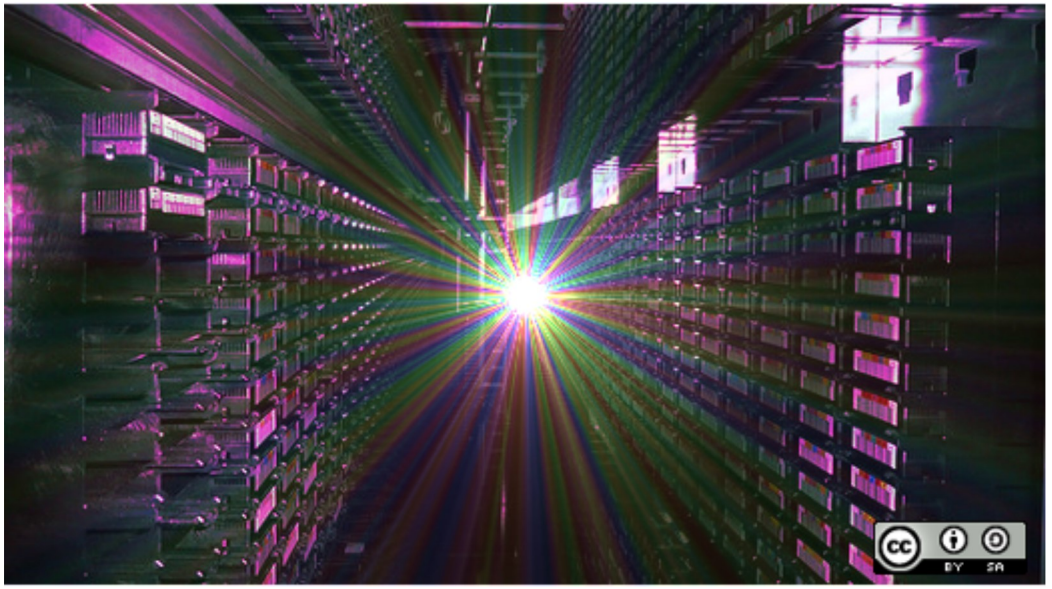


Image by : *Cory Doctorow. Modified by Opensource.com. CC BY-SA 2.0.*

Read:

- *Part 1: What is Git? (https://opensource.com/resources/what-is-git)*
- *Part 2: Getting started with Git* (/life/16/7/stumbling-git)
- *Part 3: Creating your first Git repository (https://opensource.com/life/16/7/creating-your-first-git-repository)*
- *Part 4: How to restore older file versions in Git* (/life/16/7/how-restore-older-file-versions-git)

- *Part 7: How to manage binary blobs with Git* (https://opensource.com/life/16/8/how-manage-binary-blobs-git-part-7)

Now we will learn how to build a Git server, and how to write custom Git hooks to trigger specific actions on certain events (such as notifications), and publishing your code to a website.

Up until now, the focus has been interacting with Git as a user. In this article I'll discuss the administration of Git, and the design of a flexible Git infrastructure. You might think it sounds like a euphemism for "advanced Git techniques" or "only read this if you're a super-nerd", but actually none of these tasks require advanced knowledge or any special training beyond an intermediate understanding of how Git works, and in some cases a little bit of knowledge about Linux.

Shared Git server

Creating your own shared Git server is surprisingly simple, and in many cases well worth the trouble. Not only does it ensure that you always have access to your code, it also opens doors to stretching the reach of Git with extensions such as personal Git hooks, unlimited data storage, and continuous integration and deployment.

If you know how to use Git and SSH, then you already know how to create a Git server. The way Git is designed, the moment you create or clone a repository, you have already

However, that's a little *ad hoc*. With some planning a you can construct a well-designed Git server with about the same amount of effort, but with better scalability.

First things first: identify your users, both current and in the future. If you're the only user then no changes are necessary, but if you intend to invite contributors aboard, then you should allow for a dedicated shared system user for your developers.

Assuming that you have a server available (if not, that's not exactly a problem Git can help with, but CentOS on a Raspberry Pi 3 (https://wiki.centos.org/SpecialInterestGroup/AltArch/Arm32/RaspberryPi3) is a good start), then the first step is to enable SSH logins using only SSH key authorization. This is much stronger than password logins because it is immune to brute-force attacks, and disabling a user is as simple as deleting their key.

Once you have SSH key authorization enabled, create the *gituser*. This is a shared user for all of your authorized users:

```
$ su -c 'adduser gituser'
```

Then switch over to that user, and create an *~/.ssh* framework with the appropriate permissions. This is important, because for your own protection SSH *will default to failure* if you set the permissions too liberally.

```
$ su - gituser
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys
$ chmod 600 .ssh/authorized_keys
```

The *authorized_keys* file holds the SSH public keys of all developers you give permission to work on your Git project. Your developers must create their own SSH key pairs and send you their public keys. Copy the public keys into the gituser's *authorized_keys* file. For instance, for a developer called Bob, run these commands:

```
$ cat ~/path/to/id_rsa.bob.pub >> \
/home/gituser/.ssh/authorized_keys
```

As long as developer Bob has the private key that matches the public key he sent you, Bob can access the server as gituser.

However, you don't really want to give your developers access to your server, even if only as gituser. You only want to give them access to the Git repository. For this very reason, Git provides a limited shell called, appropriately, *git-shell*. Run these commands as root to add *git-shell* to your system, and then make it the default shell for your gituser:

```
# grep git-shell /etc/shells || su -c \
"echo `which git-shell` >> /etc/shells"
# su -c 'usermod -s git-shell gituser'
```


login shell. You should add yourself to the corresponding group for the gituser, which in our example server is also gituser.

For example:

```
# usermod -a -G gituser seth
```

The only step remaining is to make a Git repository. Since no one is going to interact with it directly on the server (that is, you're not going to SSH to the server and work directly in this repository), make it a bare repository. If you want to use the repo on the server to get work done, you'll clone it from where it lives and work on it in your home directory.

Strictly speaking, you don't have to make this a bare repository; it would work as a normal repo. However, a bare repository has no *working tree* (that is, no branch is ever in a `checkout` state). This is important because remote users are not permitted to push to an active branch (how would you like it if you were working in a `dev` branch and suddenly someone pushed changes into your workspace?). Since a bare repo can have no active branch, that won't ever be an issue.

You can place this repository anywhere you please, just as long as the users and groups you want to grant permission to access it can do so. You do NOT want to store the directory in a user's home directory, for instance, because the permissions there are pretty strict, but in a common shared location, such as */opt* or */usr/local/share*.

```
# git init --bare /opt/jupiter.git
# chown -R gituser:gituser /opt/jupiter.git
# chmod -R 770 /opt/jupiter.git
```

Now any user who is either authenticated as gituser or is in the gituser group can read from and write to the `jupiter.git` repository. Try it out on a local machine:

```
$ git clone gituser@example.com:/opt/jupiter.git jupiter.clone
Cloning into 'jupiter.clone'...
Warning: you appear to have cloned an empty repository.
```

Remember: developers MUST have their public SSH key entered into the *authorized_keys* file of gituser, or if they have accounts on the server (as you would), then they must be members of the gituser group.

Git hooks

One of the nice things about running your own Git server is that it makes Git hooks available. Git hosting services sometimes provide a hook-like interface, but they don't give you true Git hooks with access to the file system. A Git hook is a script that gets executed at some point during a Git process; a hook can be executed when a repository is about to receive a commit, or after it has accepted a commit, or before it receives a push, or after a push, and so on.

standard naming scheme, is executed at the designated time. When a script should be executed is determined by the name; a `pre-push` script is executed before a push, a `post-receive` script is executed after a commit has been received, and so on. It's more or less self-documenting.

Scripts can be written in any language; if you can execute a language's `hello world` script on your system, then you can use that language to script a Git hook. By default, Git ships with some samples but does not have any enabled.

Want to see one in action? It's easy to get started. First, create a Git repository if you don't already have one:

```
$ mkdir jupiter
$ cd jupiter
$ git init .
```

Then write a "hello world" Git hook. Since I use `tcsh` at work for legacy support, I'll stick with that as my scripting language, but feel free to use your preferred language (Bash, Python, Ruby, Perl, Rust, Swift, Go) instead:

```
$ echo "#\!/bin/tcsh" > .git/hooks/post-commit
$ echo "echo 'POST-COMMIT SCRIPT TRIGGERED'" >> \
~/jupiter/.git/hooks/post-commit
$ chmod +x ~/jupiter/.git/hooks/post-commit
```


```
$ echo "hello world" > foo.txt
$ git add foo.txt
$ git commit -m 'first commit'
! POST-COMMIT SCRIPT TRIGGERED
[master (root-commit) c8678e0] first commit
1 file changed, 1 insertion(+)
create mode 100644 foo.txt
```

And there you have it: your first functioning Git hook.

The famous push-to-web hook

A popular use of Git hooks is to automatically push changes to a live, in-production web server directory. It is a great way to ditch FTP, retain full version control of what is in production, and integrate and automate publication of content.

If done correctly, it works brilliantly and is, in a way, exactly how web publishing should have been done all along. It is that good. I don't know who came up with the idea initially, but the first I heard of it was from my Emacs- and Git- mentor, Bill von Hagen at IBM. His article remains the definitive introduction to the process: Git changes the game of distributed Web development (http://www.ibm.com/developerworks/library/wa-git/).

Git variables

you want is a generic email alerting you that someone pushed *something*, then you don't need specifics, and probably don't even need to write the script as the existing samples may work for you. If you want to see the commit message and author of a commit in that email, then your script becomes more demanding.

Git hooks aren't run by the user directly, so figuring out how to gather important information can be confusing. In fact, a Git hook script is just like any other script, accepting arguments from `stdin` in the same way that BASH, Python, C++, and anything else does. The difference is, we aren't providing that input ourselves, so to use it you need to know what to expect.

Before writing a Git hook, look at the samples that Git provides in your project's *.git/hooks* directory. The *pre-push.sample* file, for instance, states in the comments section:

```
# $1 -- Name of the remote to which the push is being done
# $2 -- URL to which the push is being done
# If pushing without using a named remote those arguments will be equal.
#
# Information about commit is supplied as lines
# to the standard input in this form:
# <local ref> <local sha1> <remote ref> <remote sha1>
```

Not all samples are that clear, and documentation on what hook gets what variable is still a little sparse (unless you want to read the source code of Git), but if in doubt, you can learn

Branch detection example

I have found that a common requirement in production instances is a hook that triggers specific events based on what branch is being affected. Here is an example of how to tackle such a task.

First of all, Git hooks are not, themselves, version controlled. That is, Git doesn't track its own hooks because a Git hook is part of Git, not a part of your repository. For that reason, a Git hook that oversees commits and pushes probably make most sense living in a bare repository on your Git server, rather than as a part of your local repositories.

Let's write a hook that runs upon post-receive (that is, after a commit has been received). The first step is to identify the branch name:

```
#!/bin/tcsh

foreach arg ( $< )
  set argv = ( $arg )
  set refname = $1
end
```

This for-loop reads in the first arg (`$1`) and then loops again to overwrite that with the value of the second (`$2`), and then again with the third (`$3`). There is a better way to do that in Bash: use the `read` command and put the values into an array. However, this being tcsh and

When we have the `refname` of what is being commited, we can use Git to discover the human-readable name of the branch:

```
set branch = `git rev-parse --symbolic --abbrev-ref $refname`
echo $branch #DEBUG
```

And then compare the branch name to the keywords we want to base the action on:

```
if ( "$branch" == "master" ) then
  echo "Branch detected: master"
  git \
    --work-tree=/path/to/where/you/want/to/copy/stuff/to \
    checkout -f $branch || echo "master fail"
else if ( "$branch" == "dev" ) then
  echo "Branch detected: dev"
  Git \
    --work-tree=/path/to/where/you/want/to/copy/stuff/to \
    checkout -f $branch || echo "dev fail"
  else
    echo "Your push was successful."
    echo "Private branch detected. No action triggered."
endif
```

Make the script executable:

Now when a user commits to the server's master branch, the code is copied to an in-production directory, a commit to the dev branch get copied someplace else, and any other branch triggers no action.

It's just as simple to create a pre-commit script that, for instance, checks to see if someone is trying to push to a branch that they should not be pushing to, or to parse commit messages for **approval** strings, and so on.

Git hooks can get complex, and they can be confusing due to the level of abstraction that working through Git imposes, but they're a powerful system that allows you to design all manner of actions in your Git infrastructure. They're worth dabbling in, if only to become familiar with the process, and worth mastering if you're a serious Git user or full-time Git admin.

In our next and final article in this series, we will learn how to use Git to manage non-text binary blobs, such as audio and graphics files.

Topics : **Git** (/tags/git) **Getting started with Git** (/tags/git-series)

About the author

Seth Kenlon - Seth Kenlon is a UNIX geek, free culture advocate, independent multimedia

(/users/seth) industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project Slackermedia (http://slackermedia.info).

• More about me (/users/seth)

Recommended reading

10 things to love about Git (/article/20/12/git?utm_campaign=intrel)

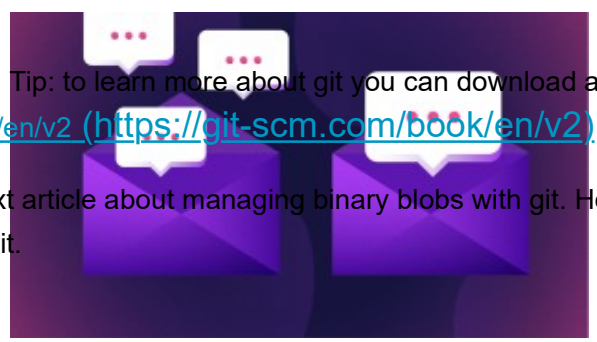
8 Git aliases that make me more efficient (/article/20/11/git-aliases?utm_campaign=intrel)

Keep track of multiple Git remote repositories (/article/20/11/multiple-git-repositories?utm_campaign=intrel)

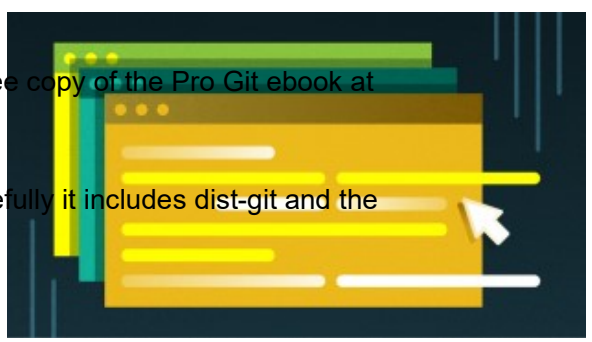
Pieter on 09 Aug 2016

Nice article Seth, thanks. Tip: to learn more about git you can download a free copy of the Pro Git ebook at https://git-scm.com/book/en/v2 (https://git-scm.com/book/en/v2)

I look forward to your next article about managing binary blobs with git. Hopefully it includes dist-git and the steps to install/configure it.



Tweak your Git config for multiple user IDs (/article/20/10/git-config?utm_campaign=intrel)



7 Git tricks that changed my life (/article/20/10/advanced-git-tips?utm_campaign=intrel)

Get started with Fossil, an alternative to Git (/article/20/11/fossil?utm_campaign=intrel)

Alan Hodgson on 09 Aug 2016

If you want to simplify this, you can install the gitolite package on your server. It greatly helps with user/key management and permissions.

kaybee on 10 Aug 2016

Great series. It has been especially timely for me as I have been setting up my own git server on a Synology NAS. Thanks for the useful series of articles!

And now to "git pick" on one thing (sorry, couldn't resist)

In the example that creates the post-commit script, I think the second line should have >> rather than > before the line continuation character so that the echo line is appended to the post-commit script rather than overwriting it. The loss of the express invokation of tcsh is pretty harmless as echo without flags works the same in pretty much any shell :-)

libreman on 10 Aug 2016

You quote:

"You do NOT want to store the directory in a user's home directory, for instance, because the permissions there are pretty strict, but in a common shared location, such as /opt or /usr/local/share."

I think /home/gituser has the right perm to do any git operation (clone, push, pull, fetch, ...) over ssh.

Seth Kenlon (/users/seth) on 10 Aug 2016

I recommend that to new admins because I've noticed some minor stumbling blocks with permissions otherwise. Nothing to do with git or ssh, so maybe I should not have mentioned it in this article; either way, I guess it all depends, in the end, with how a local system is configured. That's what makes this stuff so fun to write about: since anything is possible, everything I say is both true and false :-)

JJ (/users/wavesailor) on 15 Aug 2016

Thanks Seth for the article.
I see it is geared towards CentOS but I'm running Debian / Ubuntu on my systems. Would the commands be the same or are they different?

Thanks again

They'll be the same. There's nothing distro-specific to git.

Sachin Patil (/users/psachin) on 16 Aug 2016

Nice article Seth!

Dougie Lawson on 16 Aug 2016

Is this really any easier than using a ready built local Git server like Gitolite?
http://gitolite.com/gitolite/index.html (http://gitolite.com/gitolite/index.html)

Seth Kenlon (/users/seth) on 16 Oct 2016

Not necessarily easier, but then again sometimes a personal git server is what one needs/wants. Other times, gitolite is what one needs or wants. It's all a matter of use-case and preference.

Also, practise makes perfect, and there's no way to get to know the intricacies of git than to practise. Setting up a git server, even just for the practise, is a very useful activity.

Lewis Cowles on 27 Oct 2016

Can't ssh users override their shell so they are not using git shell?

Seth Kenlon (/users/seth) on 27 Oct 2016

I don't understand exactly what you mean, but like so much else when managing a server, a lot of what's possible depends on how you configure things. I don't know of any way to "break out" of git-shell into a Unix shell if a user has no ability to launch a remote shell on a server (it's difficult to launch /usr/bin/false, I've found). Unless you're speaking of using an exploit that I've not heard of yet...but in that case, possibilities are limitless, since we can't know about the exploits that we don't know about.