**Git on the server**

## **Bare repository**

A remote repository is generally a ***bare repository*** — a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk; it’s just the Git data. In the simplest terms, a bare repository is the contents of your project’s .git directory and nothing else.

A bare repository is the same as default, but no commits can be made in a bare repository.

The changes made in projects cannot be tracked by a bare repository as it doesn’t have a working tree.

Basically repo which has only .git then it is called bare repo.

* GitHub, GitLab, etc. store a bare repository.

No, Git repos on GitHub are bare, like any remote repo to which you want to push.

Their web interface shows you a representation of the git repo content, and they might a temporary working tree to modify files directly from the web UI (for their [inline edit](https://github.com/blog/143-inline-file-editing) introduced [initially in 2011](https://github.com/blog/905-edit-like-an-ace)), but it is their own internal mechanism.  
From a client's perspective, you can see those as classic bare repos.

**Usecase:**

A bare Git repository is typically used as a [Remote Repository](https://mijingo.com/blog/what-is-a-git-remote-repository) that is sharing a repository among several different people. You don't do work right inside the remote repository so there's no Working Tree (the files in your project that you edit), just bare repository data.

You can only push and pull.

**Bare repo clone**

$ git clone --bare <https://github.com/fvenkat/bare.git>

It will have only .git not workspace

**Bare repo initialization**

**$git init –bare**

**Non bare repo to bare repo**

* cd repo
* mv .git /desktop/new\_repo # renaming just for clarity
* cd ..
* rm -fr repo
* cd /desktop/new\_repo
* git config --bool core.bare true

## **Git protocols**

* Local, HTTP, Secure Shell (SSH) and Git.

**Local Protocol**

* The most basic is the *Local protocol*, in which the remote repository is in another directory on the same host. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer.
* The latter wouldn’t be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.
* If you have a shared mounted filesystem, then you can clone, push to, and pull from a local filebased repository.
* To clone a repository like this, or to add one as a remote to an existing project, use the path to the repository as the URL

$ git clone <path>

$ git clone /srv/git/project.git or

$ git clone <file:///srv/git/project.git>

* If you just specify the path, Git tries to use hardlinks or directly copy the files it needs.
* If you specify file://, Git fires up the processes that it normally uses to transfer data over a network, which is generally much less efficient. The main reason to specify the file:// prefix is if you want a clean copy of the repository.

**The Cons**

* The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you’re at home, you have to mount the remote disk, which can be difficult and slow compared to network based access.
* A repository on NFS is often slower than the repository over SSH on the same server, allowing Git to run off local disks on each system.

Finally, this protocol does not protect the repository against accidental damage. Every user has full shell access to the “remote” directory, and there is nothing preventing them from changing or removing internal Git files and corrupting the repository.

**Smart HTTP** [Runs on https port]

$ git clone https://example.com/gitproject.git

**The SSH Protocol**

* $ git clone ssh://[user@]server/project.git
* $ git clone [user@]server:project.git

**The Cons**

* The negative aspect of SSH is that it doesn’t support anonymous access to your Git repository. If you’re using SSH, people *must* have SSH access to your machine, even in a read-only capacity, which doesn’t make SSH conducive to open source projects for which people might simply want to clone your repository to examine it.

**Git protocol**

* it listens on a dedicated port (**9418**) that provides a service similar to the SSH protocol, but with absolutely no authentication.
* In order for a repository to be served over the Git protocol, you must create a
* git-daemon-export-ok file — the daemon won’t serve a repository without that file in it — but, other than that, there is no security.

## **Setting up git repo on own server**

In order to initially set up any Git server, you have to export an existing repository into a new bare repository — a repository that doesn’t contain a working directory

$ git clone --bare my\_project my\_project.git

**Putting the Bare Repository on a Server**

* Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let’s say you’ve set up a server called git.example.com to which you have SSH access, and you want to store all your Git repositories under the /srv/git directory. Assuming that /srv/git exists on that server, you can set up your new repository by copying your bare repository over:

$ scp -r my\_project.git [user@git.example.com:/srv/git](mailto:user@git.example.com:/srv/git)

Now other user can able to clone

$ git clone [user@git.example.com:/srv/git/my\_project.git](mailto:user@git.example.com:/srv/git/my_project.git)

* $ ssh user@git.example.com
* $ cd /srv/git/my\_project.git
* $ git init --bare --shared

**Setup User management**

If you want to place your repositories on a server that doesn’t have accounts for everyone on your team for whom you want to grant write access, then you must set up SSH access for them.

**Method one:**

The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run adduser (or the possible alternative useradd) and have to set temporary passwords for every new user.

**Method Two:**

Create a single git user account on the machine, ask every user who is to

have write access to send you an SSH public key, and add that key to the ~/.ssh/authorized\_keys file of that new git account. At that point, everyone will be able to access that machine via the git

account.

**Client END [Generating your SSH public keys]**

In order to provide a public key, each user in your system must generate one if they don’t already have one.

$ ssh-keygen

$ cd ~/.ssh

$ cat ~/.ssh/id\_rsa.pub

**Server END**

$ sudo adduser git

$ su git

$ cd

$ mkdir .ssh && chmod 700 .ssh

$ touch .ssh/authorized\_keys && chmod 600 .ssh/authorized\_keys

Next, you need to add some developer SSH public keys to the authorized\_keys file for the git user.

You just append them to the git user’s authorized\_keys file in its .ssh directory

$ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys

* You should note that currently all these users can also log into the server and get a shell as the git user. If you want to restrict that, you will have to change the shell to something else in the /etc/passwd file.
* You can easily restrict the git user account to only Git-related activities with a limited shell tool called git-shell that comes with Git. If you set this as the git user account’s login shell, then that account can’t have normal shell access to your server. To use this, specify git-shell instead of bash or csh for that account’s login shell. To do so, you must first add the full pathname of the git-shell command to /etc/shells if its not already there:

$ cat /etc/shells # see if `git-shell` is already in there. If not...

$ which git-shell # make sure git-shell is installed on your system.

$ sudo -e /etc/shells # and add the path to git-shell from last command

Now you can edit the shell for a user using chsh <username> -s <shell>:

$ sudo chsh git -s $(which git-shell)

Now, the git user can still use the SSH connection to push and pull Git repositories but can’t shell onto the machine. If you try, you’ll see a login rejection like this:

$ ssh git@gitserver

fatal: Interactive git shell is not enabled.

hint: ~/git-shell-commands should exist and have read and execute access.

Connection to gitserver closed.

At this point, users are still able to use SSH port forwarding to access any host the git server is able to reach. If you want to prevent that, you can edit the authorized\_keys file and prepend the following options to each key you’d like to restrict:

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

**Result**

$ cat ~/.ssh/authorized\_keys

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJCUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...