# Setting Up the Server

Let’s walk through setting up SSH access on the server side. In this example, you’ll use the authorized\_keys method for authenticating your users. We also assume you’re running a standard Linux distribution like Ubuntu.

|  |  |
| --- | --- |
| Note | A good deal of what is described here can be automated by using the ssh-copy-id command, rather than manually copying and installing public keys. |

First, you create a git user account and a .ssh directory for that user.

$ 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. Let’s assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

$ cat /tmp/id\_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L

ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k

Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez

Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv

O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq

dAv8JggJICUvax2T9va5 gsg-keypair

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

Now, you can set up an empty repository for them by running git init with the --bare option, which initializes the repository without a working directory:

$ cd /srv/git

$ mkdir project.git

$ cd project.git

$ git init --bare

Initialized empty Git repository in /srv/git/project.git/

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let’s use gitserver as the hostname of the server on which you’ve set up your git user and repository. If you’re running it internally, and you set up DNS for gitserver to point to that server, then you can use the commands pretty much as is (assuming that myproject is an existing project with files in it):

# on John's computer

$ cd myproject

$ git init

$ git add .

$ git commit -m 'Initial commit'

$ git remote add origin git@gitserver:/srv/git/project.git

$ git push origin master

At this point, the others can clone it down and push changes back up just as easily:

$ git clone git@gitserver:/srv/git/project.git

$ cd project

$ vim README

$ git commit -am 'Fix for README file'

$ git push origin master

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

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 it’s 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

The result should look like this:

$ cat ~/.ssh/authorized\_keys

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

AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h

PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N

YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC

IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd

LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ

ICUvax2T9va5 gsg-keypair

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

AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...

Now Git network commands will still work just fine but the users won’t be able to get a shell. As the output states, you can also set up a directory in the git user’s home directory that customizes the git-shell command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run git help shell for more information on customizing the shell.

# Git on the Server - Git Daemon

## Git Daemon

Next we’ll set up a daemon serving repositories using the “Git” protocol. This is a common choice for fast, unauthenticated access to your Git data. Remember that since this is not an authenticated service, anything you serve over this protocol is public within its network.

If you’re running this on a server outside your firewall, it should be used only for projects that are publicly visible to the world. If the server you’re running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don’t want to have to add an SSH key for each.

In any case, the Git protocol is relatively easy to set up. Basically, you need to run this command in a daemonized manner:

$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

The --reuseaddr option allows the server to restart without waiting for old connections to time out, while the --base-path option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you’re running a firewall, you’ll also need to punch a hole in it at port 9418 on the box you’re setting this up on.

You can daemonize this process a number of ways, depending on the operating system you’re running.

Since systemd is the most common init system among modern Linux distributions, you can use it for that purpose. Simply place a file in /etc/systemd/system/git-daemon.service with these contents:

[Unit]

Description=Start Git Daemon

[Service]

ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/

Restart=always

RestartSec=500ms

StandardOutput=syslog

StandardError=syslog

SyslogIdentifier=git-daemon

User=git

Group=git

[Install]

WantedBy=multi-user.target

You might have noticed that Git daemon is started here with git as both group and user. Modify it to fit your needs and make sure the provided user exists on the system. Also, check that the Git binary is indeed located at /usr/bin/git and change the path if necessary.

Finally, you’ll run systemctl enable git-daemon to automatically start the service on boot, and can start and stop the service with, respectively, systemctl start git-daemon and systemctl stop git-daemon.

On other systems, you may want to use xinetd, a script in your sysvinit system, or something else — as long as you get that command daemonized and watched somehow.

Next, you have to tell Git which repositories to allow unauthenticated Git server-based access to. You can do this in each repository by creating a file named git-daemon-export-ok.

$ cd /path/to/project.git

$ touch git-daemon-export-ok

The presence of that file tells Git that it’s OK to serve this project without authentication.

# Git on the Server - Smart HTTP

## Smart HTTP

We now have authenticated access through SSH and unauthenticated access through git://, but there is also a protocol that can do both at the same time. Setting up Smart HTTP is basically just enabling a CGI script that is provided with Git called git-http-backend on the server. This CGI will read the path and headers sent by a git fetch or git push to an HTTP URL and determine if the client can communicate over HTTP (which is true for any client since version 1.6.6). If the CGI sees that the client is smart, it will communicate smartly with it; otherwise it will fall back to the dumb behavior (so it is backward compatible for reads with older clients).

Let’s walk through a very basic setup. We’ll set this up with Apache as the CGI server. If you don’t have Apache setup, you can do so on a Linux box with something like this:

$ sudo apt-get install apache2 apache2-utils

$ a2enmod cgi alias env

This also enables the mod\_cgi, mod\_alias, and mod\_env modules, which are all needed for this to work properly.

You’ll also need to set the Unix user group of the /srv/git directories to www-data so your web server can read- and write-access the repositories, because the Apache instance running the CGI script will (by default) be running as that user:

$ chgrp -R www-data /srv/git

Next we need to add some things to the Apache configuration to run the git-http-backend as the handler for anything coming into the /git path of your web server.

SetEnv GIT\_PROJECT\_ROOT /srv/git

SetEnv GIT\_HTTP\_EXPORT\_ALL

ScriptAlias /git/ /usr/lib/git-core/git-http-backend/

If you leave out GIT\_HTTP\_EXPORT\_ALL environment variable, then Git will only serve to unauthenticated clients the repositories with the git-daemon-export-ok file in them, just like the Git daemon did.

Finally you’ll want to tell Apache to allow requests to git-http-backend and make writes be authenticated somehow, possibly with an Auth block like this:

<Files "git-http-backend">

AuthType Basic

AuthName "Git Access"

AuthUserFile /srv/git/.htpasswd

Require expr !(%{QUERY\_STRING} -strmatch '\*service=git-receive-pack\*' || %{REQUEST\_URI} =~ m#/git-receive-pack$#)

Require valid-user

</Files>

That will require you to create a .htpasswd file containing the passwords of all the valid users. Here is an example of adding a “schacon” user to the file:

$ htpasswd -c /srv/git/.htpasswd schacon

There are tons of ways to have Apache authenticate users, you’ll have to choose and implement one of them. This is just the simplest example we could come up with. You’ll also almost certainly want to set this up over SSL so all this data is encrypted.

We don’t want to go too far down the rabbit hole of Apache configuration specifics, since you could well be using a different server or have different authentication needs. The idea is that Git comes with a CGI called git-http-backend that when invoked will do all the negotiation to send and receive data over HTTP. It does not implement any authentication itself, but that can easily be controlled at the layer of the web server that invokes it. You can do this with nearly any CGI-capable web server, so go with the one that you know best.

# Git on the Server - GitWeb

## GitWeb

Now that you have basic read/write and read-only access to your project, you may want to set up a simple web-based visualizer. Git comes with a CGI script called GitWeb that is sometimes used for this.



Figure 49. The GitWeb web-based user interface

If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a lightweight web server on your system like lighttpd or webrick. On Linux machines, lighttpd is often installed, so you may be able to get it to run by typing git instaweb in your project directory. If you’re running a Mac, Leopard comes preinstalled with Ruby, so webrick may be your best bet. To start instaweb with a non-lighttpd handler, you can run it with the --httpd option.

$ git instaweb --httpd=webrick

[2009-02-21 10:02:21] INFO WEBrick 1.3.1

[2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0]

That starts up an HTTPD server on port 1234 and then automatically starts a web browser that opens on that page. It’s pretty easy on your part. When you’re done and want to shut down the server, you can run the same command with the --stop option:

$ git instaweb --httpd=webrick --stop

If you want to run the web interface on a server all the time for your team or for an open source project you’re hosting, you’ll need to set up the CGI script to be served by your normal web server. Some Linux distributions have a gitweb package that you may be able to install via apt or dnf, so you may want to try that first. We’ll walk through installing GitWeb manually very quickly. First, you need to get the Git source code, which GitWeb comes with, and generate the custom CGI script:

$ git clone git://git.kernel.org/pub/scm/git/git.git

$ cd git/

$ make GITWEB\_PROJECTROOT="/srv/git" prefix=/usr gitweb

SUBDIR gitweb

SUBDIR ../

make[2]: `GIT-VERSION-FILE' is up to date.

GEN gitweb.cgi

GEN static/gitweb.js

$ sudo cp -Rf gitweb /var/www/

Notice that you have to tell the command where to find your Git repositories with the GITWEB\_PROJECTROOT variable. Now, you need to make Apache use CGI for that script, for which you can add a VirtualHost:

<VirtualHost \*:80>

ServerName gitserver

DocumentRoot /var/www/gitweb

<Directory /var/www/gitweb>

Options +ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch

AllowOverride All

order allow,deny

Allow from all

AddHandler cgi-script cgi

DirectoryIndex gitweb.cgi

</Directory>

</VirtualHost>

Again, GitWeb can be served with any CGI or Perl capable web server; if you prefer to use something else, it shouldn’t be difficult to set up. At this point, you should be able to visit <http://gitserver/> to view your repositories online.

# Git on the Server - GitLab

## GitLab

GitWeb is pretty simplistic though. If you’re looking for a modern, fully featured Git server, there are several open source solutions out there that you can install instead. As GitLab is one of the popular ones, we’ll cover installing and using it as an example. This is harder than the GitWeb option and will require more maintenance, but it is a fully featured option.

### Installation

GitLab is a database-backed web application, so its installation is more involved than some other Git servers. Fortunately, this process is well-documented and supported. GitLab strongly recommends installing GitLab on your server via the official Omnibus GitLab package.

The other installation options are:

GitLab Helm chart, for use with Kubernetes.

Dockerized GitLab packages for use with Docker.

From the source files.

Cloud provider such as AWS, Google Cloud Platform, Azure, OpenShift and Digital Ocean.

For more information read the [GitLab Community Edition (CE) readme](https://gitlab.com/gitlab-org/gitlab-foss/-/blob/master/README.md).

### Administration

GitLab’s administration interface is accessed over the web. Simply point your browser to the hostname or IP address where GitLab is installed, and log in as the admin user. The default username is admin@local.host, and the default password is 5iveL!fe (which you must change right away). After you’ve logged in, click the “Admin area” icon in the menu at the top right.



Figure 50. The “Admin area” item in the GitLab menu

#### Users

Everybody using your GitLab server must have a user account. User accounts are quite simple, they mainly contain personal information attached to login data. Each user account has a **namespace**, which is a logical grouping of projects that belong to that user. If the user jane had a project named project, that project’s url would be <http://server/jane/project>.



Figure 51. The GitLab user administration screen

You can remove a user account in two ways: “Blocking” a user prevents them from logging into the GitLab instance, but all of the data under that user’s namespace will be preserved, and commits signed with that user’s email address will still link back to their profile.

“Destroying” a user, on the other hand, completely removes them from the database and filesystem. All projects and data in their namespace is removed, and any groups they own will also be removed. This is obviously a much more permanent and destructive action, and you will rarely need it.

#### Groups

A GitLab group is a collection of projects, along with data about how users can access those projects. Each group has a project namespace (the same way that users do), so if the group training has a project materials, its url would be <http://server/training/materials>.



Figure 52. The GitLab group administration screen

Each group is associated with a number of users, each of which has a level of permissions for the group’s projects and the group itself. These range from “Guest” (issues and chat only) to “Owner” (full control of the group, its members, and its projects). The types of permissions are too numerous to list here, but GitLab has a helpful link on the administration screen.

#### Projects

A GitLab project roughly corresponds to a single Git repository. Every project belongs to a single namespace, either a user or a group. If the project belongs to a user, the owner of the project has direct control over who has access to the project; if the project belongs to a group, the group’s user-level permissions will take effect.

Every project has a visibility level, which controls who has read access to that project’s pages and repository. If a project is Private, the project’s owner must explicitly grant access to specific users. An Internal project is visible to any logged-in user, and a Public project is visible to anyone. Note that this controls both git fetch access as well as access to the web UI for that project.

#### Hooks

GitLab includes support for hooks, both at a project or system level. For either of these, the GitLab server will perform an HTTP POST with some descriptive JSON whenever relevant events occur. This is a great way to connect your Git repositories and GitLab instance to the rest of your development automation, such as CI servers, chat rooms, or deployment tools.

### Basic Usage

The first thing you’ll want to do with GitLab is create a new project. You can do this by clicking on the “+” icon on the toolbar. You’ll be asked for the project’s name, which namespace it should belong to, and what its visibility level should be. Most of what you specify here isn’t permanent, and can be changed later through the settings interface. Click “Create Project”, and you’re done.

Once the project exists, you’ll probably want to connect it with a local Git repository. Each project is accessible over HTTPS or SSH, either of which can be used to configure a Git remote. The URLs are visible at the top of the project’s home page. For an existing local repository, this command will create a remote named gitlab to the hosted location:

$ git remote add gitlab https://server/namespace/project.git

If you don’t have a local copy of the repository, you can simply do this:

$ git clone https://server/namespace/project.git

The web UI provides access to several useful views of the repository itself. Each project’s home page shows recent activity, and links along the top will lead you to views of the project’s files and commit log.

### Working Together

The simplest way of working together on a GitLab project is by giving each user direct push access to the Git repository. You can add a user to a project by going to the “Members” section of that project’s settings, and associating the new user with an access level (the different access levels are discussed a bit in [Groups](https://git-scm.com/book/en/v2/ch00/_gitlab_groups_section)). By giving a user an access level of “Developer” or above, that user can push commits and branches directly to the repository.

Another, more decoupled way of collaboration is by using merge requests. This feature enables any user that can see a project to contribute to it in a controlled way. Users with direct access can simply create a branch, push commits to it, and open a merge request from their branch back into master or any other branch. Users who don’t have push permissions for a repository can “fork” it to create their own copy, push commits to their copy, and open a merge request from their fork back to the main project. This model allows the owner to be in full control of what goes into the repository and when, while allowing contributions from untrusted users.

Merge requests and issues are the main units of long-lived discussion in GitLab. Each merge request allows a line-by-line discussion of the proposed change (which supports a lightweight kind of code review), as well as a general overall discussion thread. Both can be assigned to users, or organized into milestones.

This section is focused mainly on the Git-related features of GitLab, but as a mature project, it provides many other features to help your team work together, such as project wikis and system maintenance tools. One benefit to GitLab is that, once the server is set up and running, you’ll rarely need to tweak a configuration file or access the server via SSH; most administration and general usage can be done through the in-browser interface.

# Git on the Server - Third Party Hosted Options

## Third Party Hosted Options

If you don’t want to go through all of the work involved in setting up your own Git server, you have several options for hosting your Git projects on an external dedicated hosting site. Doing so offers a number of advantages: a hosting site is generally quick to set up and easy to start projects on, and no server maintenance or monitoring is involved. Even if you set up and run your own server internally, you may still want to use a public hosting site for your open source code – it’s generally easier for the open source community to find and help you with.

These days, you have a huge number of hosting options to choose from, each with different advantages and disadvantages. To see an up-to-date list, check out the GitHosting page on the main Git wiki at <https://git.wiki.kernel.org/index.php/GitHosting>.

We’ll cover using GitHub in detail in [GitHub](https://git-scm.com/book/en/v2/ch00/ch06-github), as it is the largest Git host out there and you may need to interact with projects hosted on it in any case, but there are dozens more to choose from should you not want to set up your own Git server.