4.1 Git on the Server - The Protocols

At this point, you should be able to do most of the day-to-day tasks for which you'll be using Git. However, in order to do any collaboration in Git, you'll need to have a remote Git repository. Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because you can fairly easily confuse what they're working on if you're not careful. Furthermore, you want your collaborators to be able to access the repository even if your computer is offline — having a more reliable common repository is often useful. Therefore, the preferred method for collaborating with someone is to set up an intermediate repository that you both have access to, and push to and pull from that.

Running a Git server is fairly straightforward. First, you choose which protocols you want your server to support. The first section of this chapter will cover the available protocols and the pros and cons of each. The next sections will explain some typical setups using those protocols and how to get your server running with them. Last, we'll go over a few hosted options, if you don't mind hosting your code on someone else's server and don't want to go through the hassle of setting up and maintaining your own server.

If you have no interest in running your own server, you can skip to the last section of the chapter to see some options for setting up a hosted account and then move on to the next chapter, where we discuss the various ins and outs of working in a distributed source control environment.

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.

The Protocols

Git can use four distinct protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

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 file-based 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. For example, to clone a local repository, you can run something like this:

```
$ git clone /srv/git/project.git
```

Or you can do this:

```
$ git clone file:///srv/git/project.git
```

Git operates slightly differently if you explicitly specify `file://` at the beginning of the URL. 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 with extraneous references or objects left out — generally after an import from another VCS or something similar (see Git Internals for maintenance tasks). We'll use the normal path here because doing so is almost always faster.

To add a local repository to an existing Git project, you can run something like this:

```
$ git remote add local_proj /srv/git/project.git
```

Then, you can push to and pull from that remote via your new remote name `local_proj` as though you were doing so over a network.

The Pros

The pros of file-based repositories are that they're simple and they use existing file permissions and network access. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory. We'll discuss how to export a bare repository copy for this purpose in Getting Git on a Server.

This is also a nice option for quickly grabbing work from someone else's working repository. If you and a co-worker are working on the same project and they want you to check something out, running a command like `git pull /home/john/project` is often easier than them pushing to a remote server and you subsequently fetching from it.

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.

It's important to mention that this isn't necessarily the fastest option if you're using a shared mount of some kind. A local repository is fast only if you have fast access to the data. 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.

The HTTP Protocols

Git can communicate over HTTP using two different modes. Prior to Git 1.6.6, there was only one way it could do this which was very simple and generally read-only. In version 1.6.6, a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP protocol has become very popular since it's simpler for the user and smarter about how it communicates. The newer version is often referred to as the *Smart* HTTP protocol and the older way as *Dumb* HTTP. We'll cover the newer Smart HTTP protocol first.

Smart HTTP

Smart HTTP operates very similarly to the SSH or Git protocols but runs over standard HTTPS ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.

It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the `git://` protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access.

In fact, for services like GitHub, the URL you use to view the repository online (for example, https://github.com/schacon/simplegit) is the same URL you can use to clone and, if you have access, push over.

Dumb HTTP

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler *Dumb* HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of Dumb HTTP is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific `post-update` hook, and you're done (See Git Hooks). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

That's all. The `post-update` hook that comes with Git by default runs the appropriate command (`git update-server-info`) to make HTTP fetching and cloning work properly. This command is run when you push to this repository (over SSH perhaps); then, other people can clone via something like:

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

In this particular case, we're using the `/var/www/htdocs` path that is common for Apache setups, but you can use any static web server — just put the bare repository in its path. The Git data is served as basic static files (see the Git Internals chapter for details about exactly how it's served).

Generally you would either choose to run a read/write Smart HTTP server or simply have the files accessible as read-only in the Dumb manner. It's rare to run a mix of the two services.

The Pros

We'll concentrate on the pros of the Smart version of the HTTP protocol.

The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user. Being able to authenticate with a username and password is also a big advantage over SSH, since users don't have to generate SSH keys locally and upload their public key to the server before being able to interact with it. For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

You can also serve your repositories read-only over HTTPS, which means you can encrypt the content transfer; or you can go so far as to make the clients use specific signed SSL certificates.

Another nice thing is that HTTP and HTTPS are such commonly used protocols that corporate firewalls are often set up to allow traffic through their ports.

The Cons

Git over HTTPS can be a little more tricky to set up compared to SSH on some servers. Other than that, there is very little advantage that other protocols have over Smart HTTP for serving Git content.

If you're using HTTP for authenticated pushing, providing your credentials is sometimes more complicated than using keys over SSH. There are, however, several credential caching tools you can use, including Keychain access on macOS and Credential Manager on Windows, to make this pretty painless.
Read Credential Storage to see how to set up secure HTTP password caching on your system.

The SSH Protocol

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places — and if it isn't, it's easy to do. SSH is also an authenticated network protocol and, because it's ubiquitous, it's generally easy to set up and use.

To clone a Git repository over SSH, you can specify an `ssh://` URL like this:

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

Or you can use the shorter scp-like syntax for the SSH protocol:

```
$ git clone [user@]server:project.git
```

In both cases above, if you don't specify the optional username, Git assumes the user you're currently logged in as.

The Pros

The pros of using SSH are many. First, SSH is relatively easy to set up — SSH daemons are commonplace, many network admins have experience with them, and many OS distributions are set up with them or have tools to manage them. Next, access over SSH is secure — all data transfer is encrypted and authenticated. Last, like the HTTPS, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

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. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you'll have to set up SSH for you to push over but something else for others to fetch from.

The Git Protocol

Finally, we have the Git protocol. This is a special daemon that comes packaged with Git; 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. Either the Git repository is available for everyone to clone, or it isn't. This means that there is generally no pushing over this protocol. You can enable push access but, given the lack of authentication, anyone on the internet who finds your project's URL could push to that project. Suffice it to say that this is rare.

The Pros

The Git protocol is often the fastest network transfer protocol available. If you're serving a lot of traffic for a public project or serving a very large project that doesn't require user authentication for read access, it's likely that you'll want to set up a Git daemon to serve your project. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead.

The Cons

The downside of the Git protocol is the lack of authentication. It's generally undesirable for the Git protocol to be the only access to your project. Generally, you'll pair it with SSH or HTTPS access for the few developers who have push (write) access and have everyone else use `git://` for read-only access. It's also probably the most difficult protocol to set up. It must run its own daemon, which requires `xinetd` or `systemd`configuration or the like, which isn't always a walk in the park. It also requires

firewall access to port 9418, which isn’t a standard port that corporate firewalls always allow. Behind big corporate firewalls, this obscure port is commonly blocked.

prev | next