CHAPTER 4

Git on the Server

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 change 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. I'll refer to this repository as a *Git server*; but you'll notice that it generally takes a tiny amount of resources to host a Git repo, so you'll rarely need to use an entire server for it.

Running a Git server is simple. First, you choose which protocols you want your server to communicate with. 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, I'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 I 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 major network protocols to transfer data: Local, Secure Shell (SSH), Git, and HTTP. Here I'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

It's important to note that with the exception of the HTTP protocols, all of these require Git to be installed and working on the server.

Local Protocol

The most basic is the Local protocol, in which the remote repository is in another directory on disk. 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 /opt/git/project.git
```

Or you can do this:

```
$ git clone file:///opt/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 a lot less efficient method of transferring the data. 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 version-control system or something similar (see Chapter 9 for maintenance tasks). You'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 /opt/git/project.git
```

Then, you can push to and pull from that remote 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. I'll discuss how to export a bare repository copy for this purpose in the next section, "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 pulling down.

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 also 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.

The SSH Protocol

Probably the most common transport protocol for Git is 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 the only network-based protocol that you can easily read from and write to. The other two network protocols (HTTP and Git) are generally read-only, so even if you have them available for the unwashed masses, you still need SSH for your own write commands. 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 `ssh://` URL like this:

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

Or you can not specify a protocol—Git assumes SSH if you aren't explicit:

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

You can also not specify a user, and Git assumes the user you're currently logged in as.

The Pros

The pros of using SSH are many. First, you basically have to use it if you want authenticated write access to your repository over a network. Second, 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 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 you can't serve anonymous access of your repository over it. People must have access to your machine over SSH to access it, even in a read-only capacity, which doesn't make SSH access conducive to open source projects. 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, you'll have to set up SSH for you to push over but something else for others to pull over.

The Git Protocol

Next is 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 the `git-export-daemon-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, if you turn on push access, anyone on the Internet who finds your project's URL could push to your project. Suffice it to say that this is rare.

The Pros

The Git protocol is the fastest 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 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 is custom—you'll look at setting one up in the "Gitosis" section of this chapter—and it requires xinetd 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.

The HTTP/S Protocol

Last you have the HTTP protocol. The beauty of the HTTP or HTTPS protocol is the simplicity of setting it up. Basically, all you have to do is put the bare Git repository under your HTTP document root and set up a specific post-receive hook, and you're done (See Chapter 7 for details on 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; then, other people can clone via something like

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

In this particular case, you'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 Chapter 9 for details about exactly how it's served).

It's possible to make Git push over HTTP as well, although that technique isn't as widely used and requires you to set up complex WebDAV requirements. Because it's rarely used, I won't cover it in this book. If you're interested in using the HTTP-push protocols, you can read about preparing a repository for this purpose at `http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt`. One nice thing about making Git push over HTTP is that you can use any WebDAV server, without specific Git features; so, you can use this functionality if your web-hosting provider supports WebDAV for writing updates to your web site.

The Pros

The upside of using the HTTP protocol is that it's easy to set up. Running the handful of required commands gives you a simple way to give the world read access to your Git repository. It takes only a few minutes to do. The HTTP protocol also isn't very resource intensive on your server. Because it generally uses a static HTTP server to serve all the data, a normal Apache server can serve thousands of files per second on average—it's difficult to overload even a small server.

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. Generally, if you're going to these lengths, it's easier to use SSH public keys; but it may be a better solution in your specific case to use signed SSL certificates or other HTTP-based authentication methods for read-only access over HTTPS.

Another nice thing is that HTTP is such a commonly used protocol that corporate firewalls are often set up to allow traffic through this port.

The Cons

The downside of serving your repository over HTTP is that it's relatively inefficient for the client. It generally takes a lot longer to clone or fetch from the repository, and you often have a lot more network overhead and transfer volume over HTTP than with any of the other network protocols. Because it's not as intelligent about transferring only the data you need—there is no dynamic work on the part of the server in these transactions—the HTTP protocol is often referred to as a *dumb* protocol. For more information about the differences in efficiency between the HTTP protocol and the other protocols, see Chapter 9.

Getting Git on a 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. This is generally straightforward to do.

In order to clone your repository to create a new bare repository, you run the `clone` command with the `--bare` option. By convention, bare repository directories end in `.git`, like so:

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

The output for this command is a little confusing. Because `clone` is basically a `git init` and then a `git fetch`, you see some output from the `git init` part, which creates an empty directory. The actual object transfer gives no output, but it does happen. You should now have a copy of the Git directory data in your `my_project.git` directory.

This is roughly equivalent to something like

```
$ cp -Rf my_project/.git my_project.git
```

There are a couple of minor differences in the configuration file; but for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

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` that you have SSH access to, and you want to store all your Git repositories under the `/opt/git` directory. You can set up your new repository by copying your bare repository over:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

At this point, other users who have SSH access to the same server, which has read access to the `/opt/git` directory, can clone your repository by running

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

If a user SSHs into a server and has write access to the `/opt/git/my_project.git` directory, they also automatically have push access. Git automatically adds group write permissions to a repository properly if you run the `git init` command with the `--shared` option:

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which you and your collaborators have SSH access. Now you're ready to collaborate on the same project.

It's important to note that this is literally all you need to do to run a useful Git server to which several people have access—just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You're ready to go—nothing else is needed.

In the next few sections, you'll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs, using the Gitosis tool, and more. However, keep in mind that to collaborate with a couple of people on a private project, all you need is an SSH server and a bare repository.

Small Setups

If you're a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only to certain users and read/write to others, access and permissions can be a bit difficult to arrange.

SSH Access

If you already have a server to which all your developers have SSH access, it's generally easiest to set up your first repository there, because you have to do almost no work (as I covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of the operating system your server runs.

If you want to place your repositories on a server that doesn't have accounts for everyone on your team whom you want to have write access, then you must set up SSH access for them. I assume that if you have a server with which to do this, you already have an SSH server installed, and that's how you're accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run `adduser` and set temporary passwords for every user.

A second method is to create a single "git" user 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 your new "git" user. At that point, everyone will be able to access that machine via the "git" user. This doesn't affect the commit data in any way—the SSH user you connect as doesn't affect the commits you've recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

Generating Your SSH Public Key

That being said, many Git servers authenticate using 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. This process is similar across all operating systems.

First, you should check to make sure you don't already have a key. By default, a user's SSH keys are stored in that user's `~/.ssh` directory. You can easily check to see if you have a key already by going to that directory and listing the contents:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

You're looking for a pair of files named *something* and *something*.pub, where the *something* is usually id_dsa or id_rsa. The .pub file is your public key, and the other file is your private key. If you don't have these files (or you don't even have a .ssh directory), you can create them by running a program called ssh-keygen, which is provided with the SSH package on Linux/Mac systems and comes with the MSysGit package on Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

First it confirms where you want to save the key (.ssh/id_rsa), and then it asks twice for a passphrase, which you can leave empty if you don't want to type a password when you use the key.

Now, each user that does this has to send their public key to you or whoever is administrating the Git server (assuming you're using an SSH server setup that requires public keys). All they have to do is copy the contents of the .pub file and e-mail it. The public keys look something like this:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

For a more in-depth tutorial on creating an SSH key on multiple operating systems, see the GitHub guide on SSH keys at http://github.com/guides/providing-your-ssh-key.

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. I also assume you're running a standard Linux distribution like Ubuntu. First, you create a "git" user and a .ssh directory for that user:

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Next, you need to add some developer SSH public keys to the `authorized_keys` file for that user. Let's assume you've received a few keys by e-mail and 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 append them to your `authorized_keys` file:

```
$ 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 /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

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:

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/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:/opt/git/project.git
$ vim README
$ git commit -am 'fix for the 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.

As an extra precaution, you can easily restrict the "git" user to only doing Git activities with a limited shell tool called `git-shell` that comes with Git. If you set this as your "git" user's login shell, then the "git" user can't have normal shell access to your server. To use this, specify `git-shell` instead of `bash` or `csh` for your user's login shell. To do so, you'll likely have to edit your `/etc/passwd` file:

```
$ sudo vim /etc/passwd
```

At the bottom, you should find a line that looks something like this:

```
git:x:1000:1000::/home/git:/bin/sh
```

Change `/bin/sh` to `/usr/bin/git-shell` (or run `which git-shell` to see where it's installed). The line should look something like this:

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

Now, the "git" user can only use the SSH connection to push and pull Git repositories and can't shell onto the machine. If you try, you'll see a login rejection:

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

Public Access

What if you want anonymous read access to your project? Perhaps instead of hosting an internal private project, you want to host an open source project. Or maybe you have a bunch of automated build servers or continuous integration servers that change a lot, and you don't want to have to generate SSH keys all the time—you just want to add simple anonymous read access.

Probably the simplest way for smaller setups is to run a static web server with its document root where your Git repositories are, and then enable that post-update hook I mentioned in the first section of this chapter. You'll work from the previous example. Say you have your repositories in the `/opt/git` directory, and an Apache server is running on your machine. Again, you can use any web server for this; but as an example, I'll demonstrate some basic Apache configurations that should give you an idea of what you might need.

First you need to enable the hook:

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

If you're using a version of Git earlier than 1.6, the `mv` command isn't necessary—Git started naming the hooks examples with the `.sample` postfix only recently.

What does this post-update hook do? It looks basically like this:

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

This means that when you push to the server via SSH, Git runs this command to update the files needed for HTTP fetching.

Next, you need to add a VirtualHost entry to your Apache configuration with the document root as the root directory of your Git projects. Here, I'm assuming that you have wildcard DNS set up to send `*.gitserver` to whatever box you're using to run all this:

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

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

```
$ chgrp -R www-data /opt/git
```

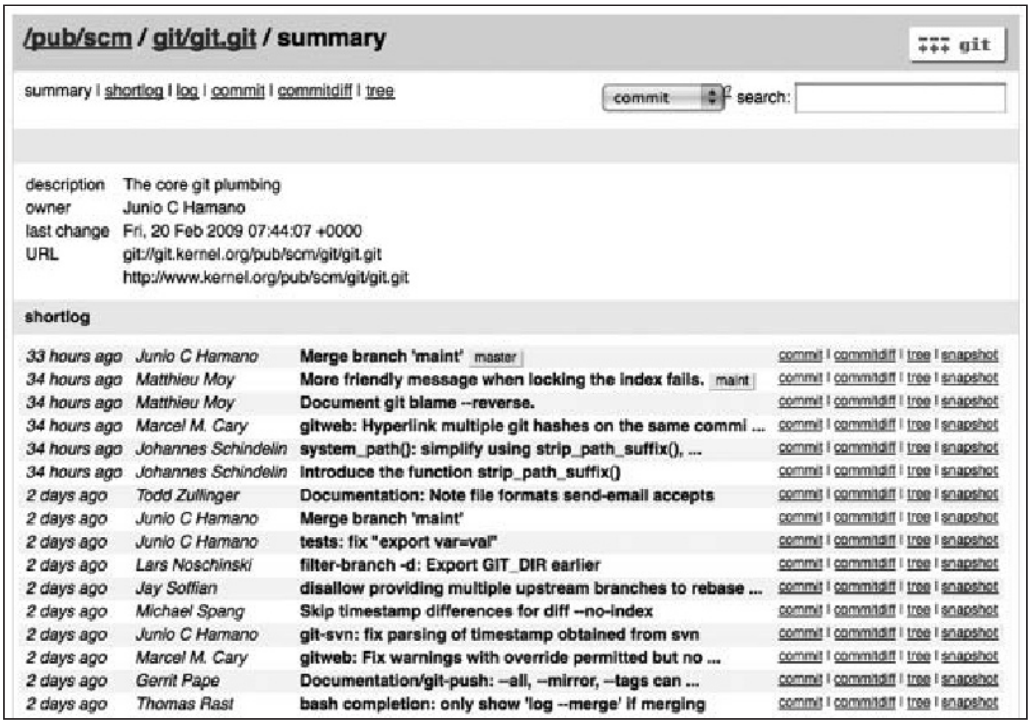
When you restart Apache, you should be able to clone your repositories under that directory by specifying the URL for your project:

```
$ git clone http://git.gitserver/project.git
```

This way, you can set up HTTP-based read access to any of your projects for a fair number of users in a few minutes. Another simple option for public unauthenticated access is to start a Git daemon, although that requires you to daemonize the process—I'll cover this option in the next section, if you prefer that route.

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 commonly used for this. You can see GitWeb in use at sites like `http://git.kernel.org` (see Figure 4-1).


/pub/scm / git/git.git / summary
git
summary | shortlog | log | commit | commitdiff | tree
commit
search:

description	The core git plumbing
owner	Junio C Hamano
last change	Fri, 20 Feb 2009 07:44:07 +0000
URL	git://git.kernel.org/pub/scm/git/git.git
	http://www.kernel.org/pub/scm/git/git.git

shortlog

33 hours ago	Junio C Hamano	Merge branch 'maint' master	commit | commitdiff | tree | snapshot
34 hours ago	Matthieu Moy	More friendly message when locking the index fails. maint	commit | commitdiff | tree | snapshot
34 hours ago	Matthieu Moy	Document git blame --reverse.	commit | commitdiff | tree | snapshot
34 hours ago	Marcel M. Cary	gitweb: Hyperlink multiple git hashes on the same commi ...	commit | commitdiff | tree | snapshot
34 hours ago	Johannes Schindelin	system_path(): simplify using strip_path_suffix(), ...	commit | commitdiff | tree | snapshot
34 hours ago	Johannes Schindelin	Introduce the function strip_path_suffix()	commit | commitdiff | tree | snapshot
2 days ago	Todd Zullinger	Documentation: Note file formats send-email accepts	commit | commitdiff | tree | snapshot
2 days ago	Junio C Hamano	Merge branch 'maint'	commit | commitdiff | tree | snapshot
2 days ago	Junio C Hamano	tests: fix "export var=val"	commit | commitdiff | tree | snapshot
2 days ago	Lars Noschinski	filter-branch -d: Export GIT_DIR earlier	commit | commitdiff | tree | snapshot
2 days ago	Jay Soffian	disallow providing multiple upstream branches to rebase ...	commit | commitdiff | tree | snapshot
2 days ago	Michael Spang	Skip timestamp differences for diff --no-index	commit | commitdiff | tree | snapshot
2 days ago	Junio C Hamano	git-svn: fix parsing of timestamp obtained from svn	commit | commitdiff | tree | snapshot
2 days ago	Marcel M. Cary	gitweb: Fix warnings with override permitted but no ...	commit | commitdiff | tree | snapshot
2 days ago	Gerrit Pape	Documentation/git-push: --all, --mirror, --tags can ...	commit | commitdiff | tree | snapshot
2 days ago	Thomas Rast	bash completion: only show 'log --merge' if merging	commit | commitdiff | tree | snapshot

Figure 4-1. *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 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 `yum`, so you may want to try that first.

I'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="/opt/git" \
        prefix=/usr gitweb/gitweb.cgi
$ 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-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, and you can use `http://git.gitserver` to clone and fetch your repositories over HTTP.

Gitosis

Keeping all users' public keys in the `authorized_keys` file for access works well only for a while. When you have hundreds of users, it's much more of a pain to manage that process. You have to shell onto the server each time, and there is no access control—everyone in the file has read and write access to every project.

You may want to turn to a widely used software project called Gitosis. Gitosis is basically a set of scripts that help you manage the `authorized_keys` file as well as implement some simple access controls. The really interesting part is that the UI for this tool for adding people and determining access isn't a web interface but a special Git repository. You set up the information in that project; and when you push it, Gitosis reconfigures the server based on that, which is cool.

Installing Gitosis isn't the simplest task ever, but it's not too difficult. It's easiest to use a Linux server for it—these examples use a stock Ubuntu 8.10 server.

Gitosis requires some Python tools, so first you have to install the Python setuptools package, which Ubuntu provides as python-setuptools:

```
$ apt-get install python-setuptools
```

Next, you clone and install Gitosis from the project's main site:

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

That installs a couple of executables that Gitosis will use. Next, Gitosis wants to put its repositories under /home/git, which is fine. But you have already set up your repositories in /opt/git, so instead of reconfiguring everything, you create a symlink:

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis is going to manage your keys for you, so you need to remove the current file, re-add the keys later, and let Gitosis control the authorized_keys file automatically. For now, move the authorized_keys file out of the way:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

You need to turn your shell back on for the "git" user, if you changed it to the git-shell command. People still won't be able to log in, but Gitosis will control that for you. So, change this line in your /etc/passwd file

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

back to this:

```
git:x:1000:1000::/home/git:/bin/sh
```

Now it's time to initialize Gitosis. You do this by running the gitosis-init command with your personal public key. If your public key isn't on the server, you'll have to copy it there:

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

This lets the user with that key modify the main Git repository that controls the Gitosis setup. Next, you have to manually set the execute bit on the post-update script for your new control repository.

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

You're ready to roll. If you're set up correctly, you can try to SSH into your server as the user for which you added the public key to initialize Gitosis. You should see something like this:

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
  Connection to gitserver closed.
```

That means Gitosis recognized you but shut you out because you're not trying to do any Git commands. So, do an actual Git command and clone the Gitosis control repository:

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

Now you have a directory named `gitosis-admin`, which has two major parts:

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

The `gitosis.conf` file is the control file you use to specify users, repositories, and permissions. The `keydir` directory is where you store the public keys of all the users who have any sort of access to your repositories—one file per user. The name of the file in `keydir` (in the previous example, `scott.pub`) will be different for you—Gitosis takes that name from the description at the end of the public key that was imported with the `gitosis-init` script.

If you look at the `gitosis.conf` file, it should only specify information about the gitosis-admin project that you just cloned:

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

It shows you that the "scott" user—the user with whose public key you initialized Gitosis—is the only one who has access to the gitosis-admin project.

Now you can add a new project. You'll add a new section called `mobile` where you'll list the developers on your mobile team and projects that those developers need access to. Because "scott" is the only user in the system right now, you add him as the only member and create a new project called iphone_project to start on:

```
[group mobile]
writable = iphone_project
members = scott
```

Whenever you make changes to the gitosis-admin project, you have to commit the changes and push them back up to the server in order for them to take effect:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
```

```
To git@gitserver:/opt/git/gitosis-admin.git
   fb27aec..8962da8  master -> master
```

You can make your first push to the new iphone_project project by adding your server as a remote to your local version of the project and pushing. You no longer have to manually create a bare repository for new projects on the server—Gitosis creates them automatically when it sees the first push:

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

Notice that you don't need to specify the path (in fact, doing so won't work), just a colon and then the name of the project—Gitosis finds it for you.

You want to work on this project with your friends, so you have to re-add their public keys. But instead of appending them manually to the `~/.ssh/authorized_keys` file on your server, you'll add them, one key per file, into the `keydir` directory. How you name the keys determines how you refer to the users in the `gitosis.conf` file. Re-add the public keys for John, Josie, and Jessica:

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Now you can add them all to your "mobile" team so they have read and write access to iphone_project:

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

After you commit and push that change, all four users will be able to read from and write to that project.

Gitosis has simple access controls as well. If you want John to have only read access to this project, you can do this instead:

```
[group mobile]
writable = iphone_project
members = scott josie jessica

[group mobile_ro]
readable = iphone_project
members = john
```

Now John can clone the project and get updates, but Gitosis won't allow him to push back up to the project. You can create as many of these groups as you want, each containing different users and projects. You can also specify another group as one of the members, to inherit all of its members automatically.

If you have any issues, it may be useful to add `loglevel=DEBUG` under the `[gitosis]` section. If you've lost push access by pushing a messed-up configuration, you can manually fix the file on the server under `/home/git/.gitosis.conf`—the file from which Gitosis reads its info. A push to the project takes the `gitosis.conf` file you just pushed up and sticks it there. If you edit that file manually, it remains like that until the next successful push to the gitosis-admin project.

Git Daemon

For public, unauthenticated read access to your projects, you'll want to move past the HTTP protocol and start using the Git protocol. The main reason is speed. The Git protocol is far more efficient and thus faster than the HTTP protocol, so using it will save your users time.

Again, this is for unauthenticated read-only access. If you're running this on a server outside your firewall, it should only be used 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=/opt/git/ /opt/git/
```

`--reuseaddr` allows the server to restart without waiting for old connections to time out, 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 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. On an Ubuntu machine, you use an Upstart script. So, in the following file

```
/etc/event.d/local-git-daemon
```

you put this script:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

For security reasons, you're strongly encouraged to have this daemon run as a user with read-only permissions to the repositories—you can easily do this by creating a new user `git-ro` and running the daemon as that user. For the sake of simplicity, run it as the same "git" user that Gitosis is running as.

When you restart your machine, your Git daemon starts automatically and respawns if it goes down. To get it running without having to reboot, you can run this:

```
initctl start local-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 your Gitosis server which repositories to allow unauthenticated Git server-based access to. If you add a section for each repository, you can specify the ones from which you want your Git daemon to allow reading. If you want to allow Git protocol access for your iphone project, you add this to the end of the `gitosis.conf` file:

```
[repo iphone_project]
daemon = yes
```

When that is committed and pushed up, your running daemon should start serving requests for the project to anyone who has access to port 9418 on your server.

If you decide not to use Gitosis, but you want to set up a Git daemon, you have to run this on each project you want the Git daemon to serve:

```
$ 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.

Gitosis can also control which projects GitWeb shows. First, you need to add something like the following to the `/etc/gitweb.conf` file:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

You can control which projects GitWeb lets users browse by adding or removing a `gitweb` setting in the Gitosis configuration file. For instance, if you want the iphone project to show up on GitWeb, you make the repo setting look like this:

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

Now, if you commit and push the project, GitWeb will automatically start showing your iphone project.

Hosted Git

If you don't want to go through all 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:

```
http://git.or.cz/gitwiki/GitHosting
```

Because I can't cover all the hosting sites, and because I happen to work at one of them, I'll use this section to walk through setting up an account and creating a new project at GitHub. This will give you an idea of what is involved.

GitHub is by far the largest open source Git hosting site, and it's also one of the very few that offers both public and private hosting options so you can keep your open source and private commercial code in the same place. In fact, I used GitHub while writing this book.

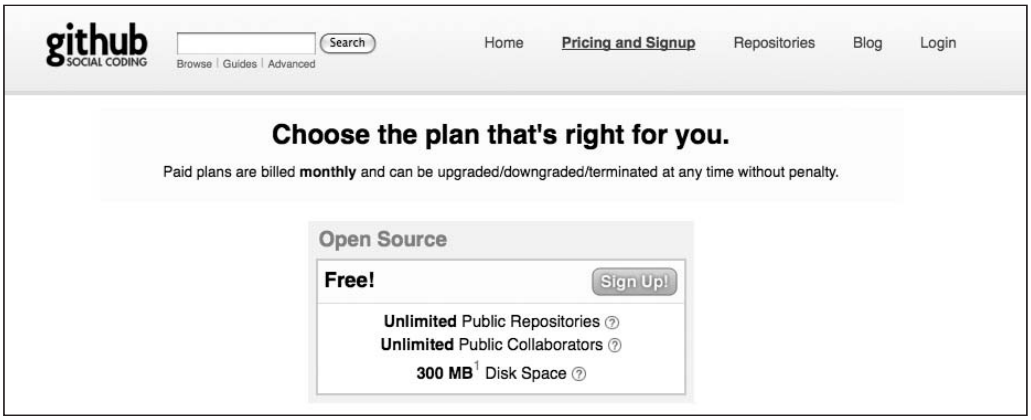
GitHub

GitHub is slightly different than most code-hosting sites in the way that it namespaces projects. Instead of being primarily based on the project, GitHub is user centric. That means when you host our grit project on GitHub, you won't find it at `github.com/grit` but instead at `github.com/schacon/grit`. There is no canonical version of any project, which allows a project to move from one user to another seamlessly if the first author abandons the project.

GitHub is also a commercial company that charges for accounts that maintain private repositories, but anyone can quickly get a free account to host as many open source projects as they want. I'll quickly go over how that is done.

Setting Up a User Account

The first thing you need to do is set up a free user account. If you visit the Pricing and Signup page at `http://github.com/plans` and click the Sign Up button on the Free account (see Figure 4-2), you're taken to the signup page.



Choose the plan that's right for you.

Paid plans are billed **monthly** and can be upgraded/downgraded/terminated at any time without penalty.

Open Source

Free! Sign Up!

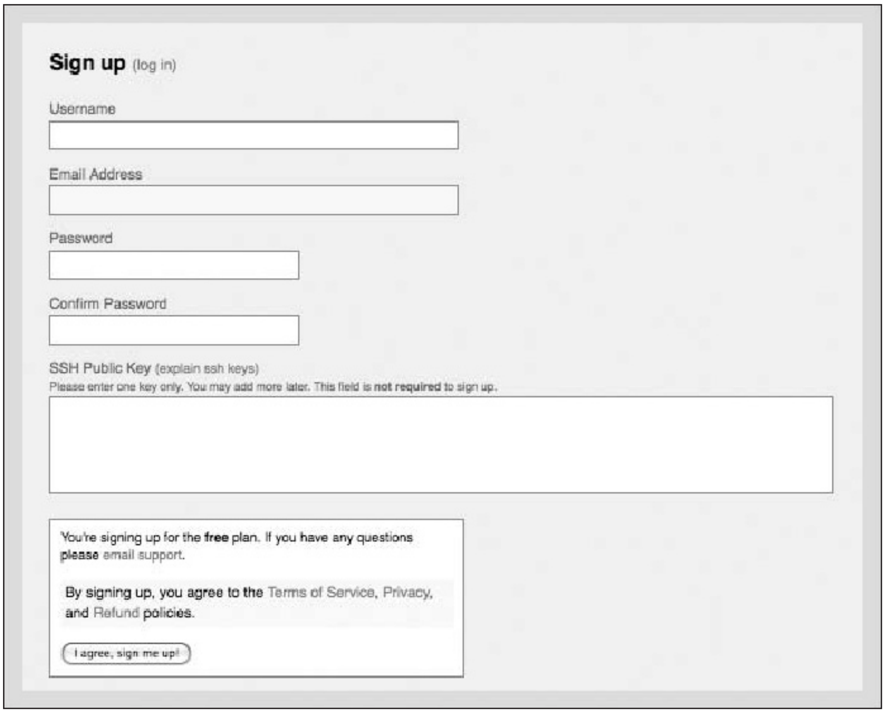
Unlimited Public Repositories

Unlimited Public Collaborators

300 MB[1] Disk Space

Figure 4-2. *The GitHub plan page*

Here you must choose a username that isn't yet taken in the system and enter an e-mail address that will be associated with the account and a password (see Figure 4-3).



Sign up (log in)

Username

Email Address

Password

Confirm Password

SSH Public Key (explain ssh keys)
Please enter one key only. You may add more later. This field is **not required** to sign up.

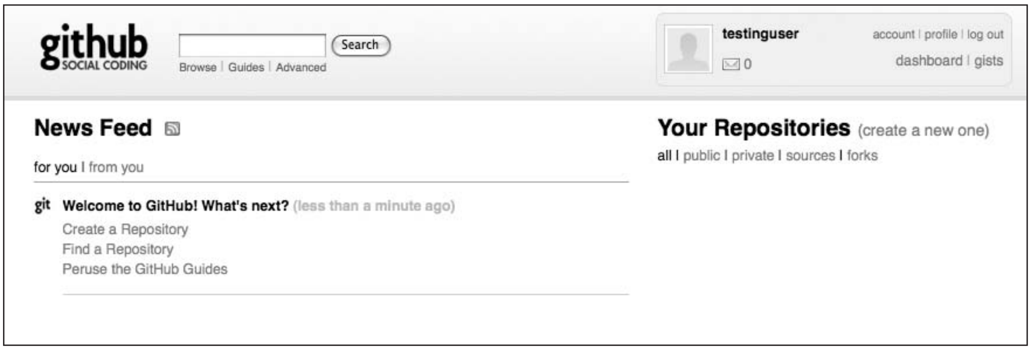
You're signing up for the **free** plan. If you have any questions please email support.

By signing up, you agree to the Terms of Service, Privacy, and Refund policies.

Figure 4-3. *The GitHub user signup form*

If you have it available, this is a good time to add your public SSH key as well. I covered how to generate a new key earlier, in the "Small Setups" section. Take the contents of the public key of that pair, and paste it into the SSH Public Key text box. Clicking the "explain ssh keys" link takes you to detailed instructions on how to do so on all major operating systems.

Clicking the "I agree, sign me up" button takes you to your new user dashboard (see Figure 4-4).



News Feed

for you | from you

git **Welcome to GitHub! What's next?** (less than a minute ago)
Create a Repository
Find a Repository
Peruse the GitHub Guides

Your Repositories (create a new one)

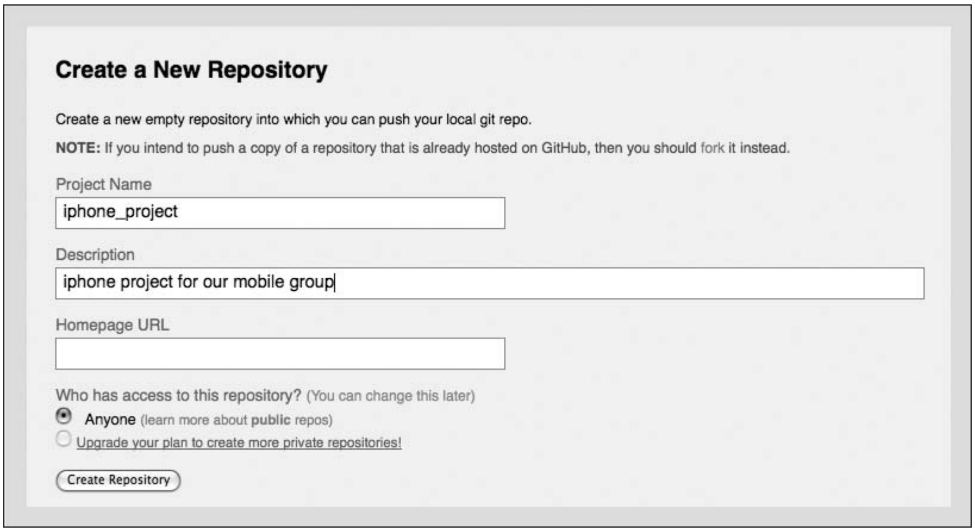
all | public | private | sources | forks

Figure 4-4. *The GitHub user dashboard*

Next, you can create a new repository.

Creating a New Repository

Start by clicking the "create a new one" link next to Your Repositories on the user dashboard. You're taken to the Create a New Repository form (see Figure 4-5).



Create a New Repository

Create a new empty repository into which you can push your local git repo.

NOTE: If you intend to push a copy of a repository that is already hosted on GitHub, then you should fork it instead.

Project Name: iphone_project

Description: iphone project for our mobile group

Homepage URL:

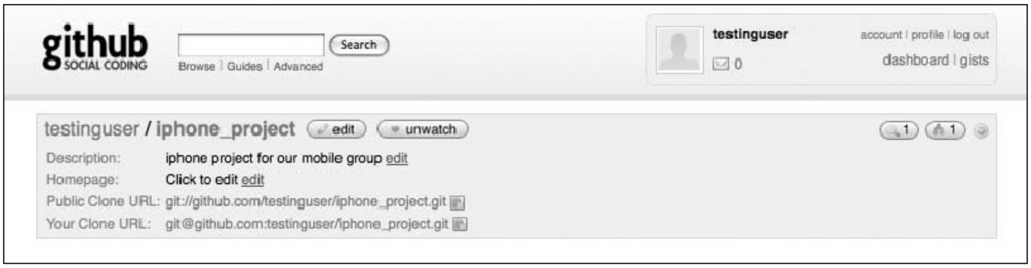
Who has access to this repository? (You can change this later)

[✓ Anyone] (learn more about **public** repos)

Upgrade your plan to create more private repositories!

Figure 4-5. *Creating a new repository on GitHub*

All you really have to do is provide a project name, but you can also add a description. When that is done, click the Create Repository button. Now you have a new repository on GitHub (see Figure 4-6).



testinguser / iphone_project

Description: iphone project for our mobile group edit

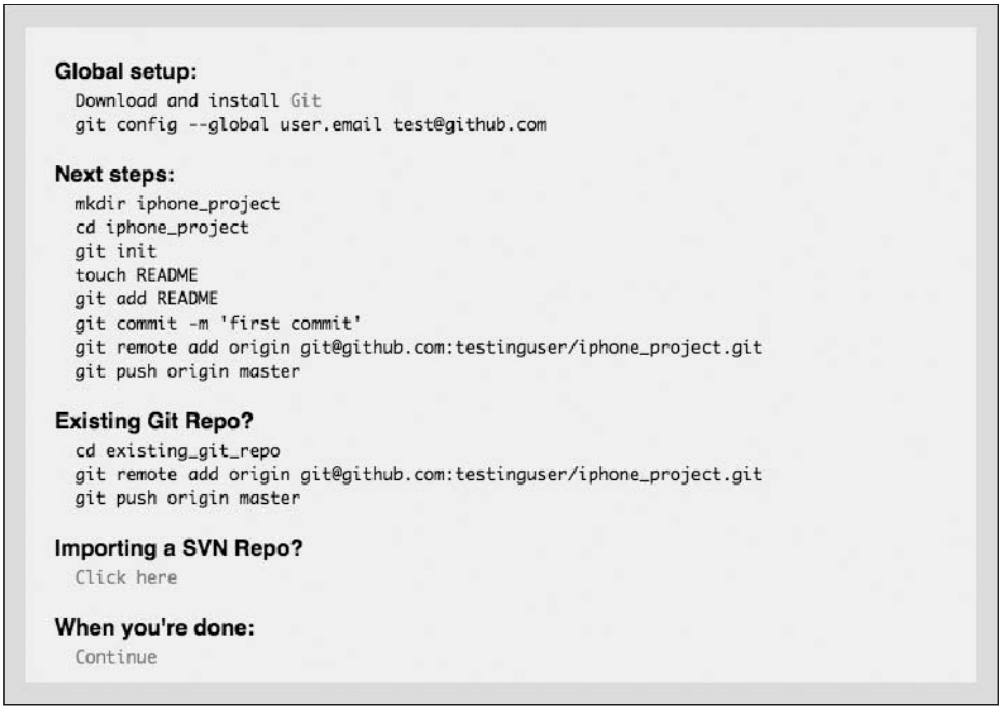
Homepage: Click to edit edit

Public Clone URL: git://github.com/testinguser/iphone_project.git

Your Clone URL: git@github.com:testinguser/iphone_project.git

Figure 4-6. *GitHub project header information*

Because you have no code there yet, GitHub shows you instructions for how to create a brand-new project, push up an existing Git project, or import a project from a public Subversion repository (see Figure 4-7).



Global setup:

```
Download and install Git
git config --global user.email test@github.com
```

Next steps:

```
mkdir iphone_project
cd iphone_project
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@github.com:testinguser/iphone_project.git
git push origin master
```

Existing Git Repo?

```
cd existing_git_repo
git remote add origin git@github.com:testinguser/iphone_project.git
git push origin master
```

Importing a SVN Repo?

Click here

When you're done:

Continue

Figure 4-7. *Instructions for a new repository*

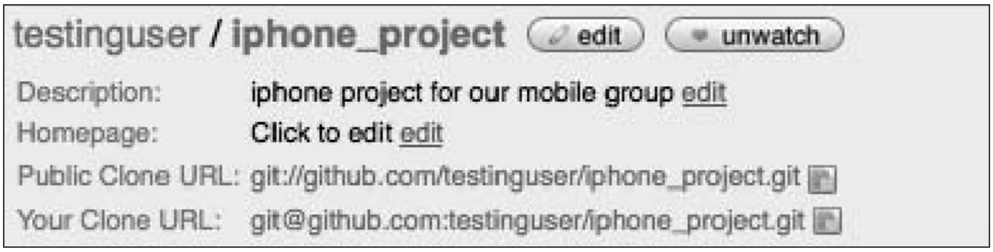
These instructions are similar to what you've already gone over. To initialize a project if it isn't already a Git project, you use

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

When you have a Git repository locally, add GitHub as a remote and push up your master branch:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Now your project is hosted on GitHub, and you can give the URL to anyone you want to share your project with. In this case, it's `http://github.com/testinguser/iphone_project`. You can also see from the header on each of your project's pages that you have two Git URLs (see Figure 4-8).


testinguser / iphone_project

Description:	iphone project for our mobile group edit
Homepage:	Click to edit edit
Public Clone URL:	git://github.com/testinguser/iphone_project.git
Your Clone URL:	git@github.com:testinguser/iphone_project.git

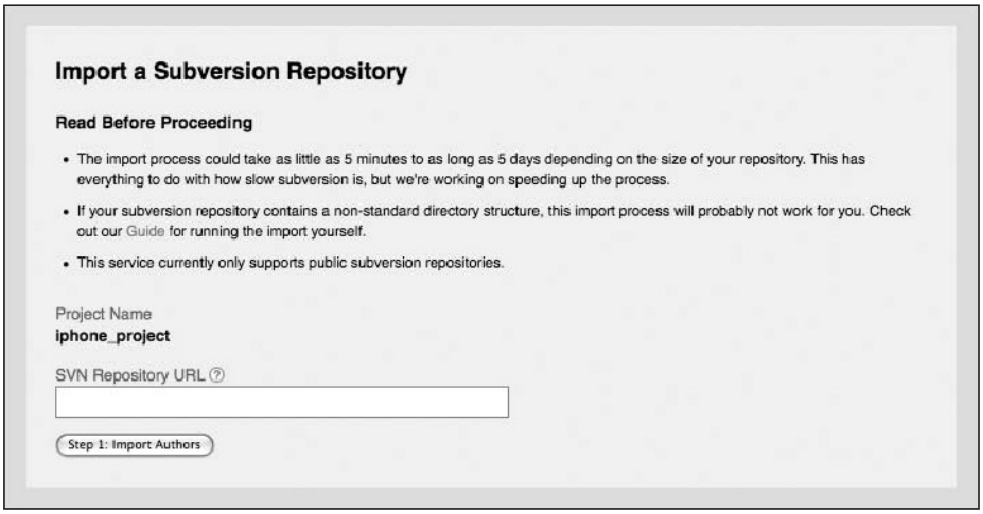
Figure 4-8. *Project header with a public URL and a private URL*

The Public Clone URL is a public, read-only Git URL over which anyone can clone the project. Feel free to give out that URL and post it on your web site or what have you.

The Your Clone URL is a read/write SSH-based URL that you can read or write over only if you connect with the SSH private key associated with the public key you uploaded for your user. When other users visit this project page, they won't see that URL—only the public one.

Importing from Subversion

If you have an existing public Subversion project that you want to import into Git, GitHub can often do that for you. At the bottom of the instructions page is a link to a Subversion import. If you click it, you see a form with information about the import process and a text box where you can paste in the URL of your public Subversion project (see Figure 4-9).


Import a Subversion Repository

Read Before Proceeding

- The import process could take as little as 5 minutes to as long as 5 days depending on the size of your repository. This has everything to do with how slow subversion is, but we're working on speeding up the process.
- If your subversion repository contains a non-standard directory structure, this import process will probably not work for you. Check out our Guide for running the import yourself.
- This service currently only supports public subversion repositories.

Project Name
iphone_project

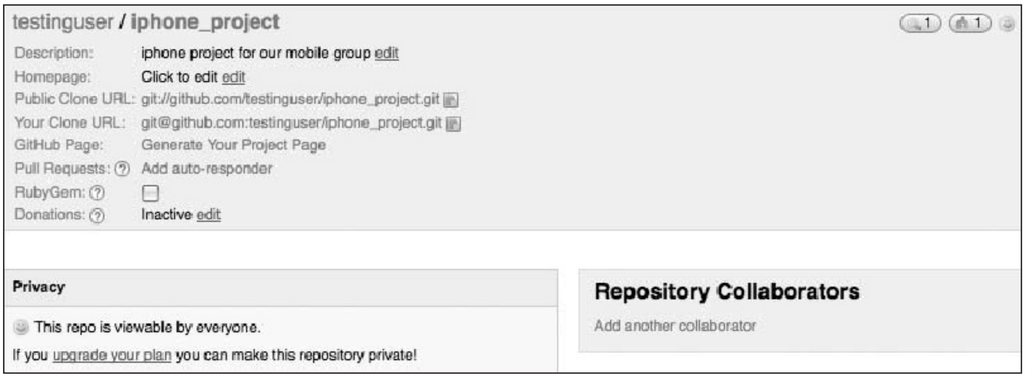
Figure 4-9. *Subversion importing interface*

If your project is very large, nonstandard, or private, this process probably won't work for you. In Chapter 7, you'll learn how to do more complicated manual project imports.

Adding Collaborators

You'll now add the rest of the team. If John, Josie, and Jessica all sign up for accounts on GitHub, and you want to give them push access to your repository, you can add them to your project as collaborators. Doing so allows pushes from their public keys to work.

Click the "edit" button in the project header or the Admin tab at the top of the project to reach the Admin page of your GitHub project (see Figure 4-10).



testinguser / iphone_project

Description:	iphone project for our mobile group edit
Homepage:	Click to edit edit
Public Clone URL:	git://github.com/testinguser/iphone_project.git
Your Clone URL:	git@github.com:testinguser/iphone_project.git
GitHub Page:	Generate Your Project Page
Pull Requests:	Add auto-responder
RubyGem:	☐
Donations:	Inactive edit

Privacy

This repo is viewable by everyone.

If you upgrade your plan you can make this repository private!

Repository Collaborators

Add another collaborator

Figure 4-10. *GitHub administration page*

To give another user write access to your project, click the "Add another collaborator" link. A new text box appears, into which you can type a username. As you type, a helper pops up, showing you possible username matches. When you find the correct user, click the Add button to add that user as a collaborator on your project (see Figure 4-11).

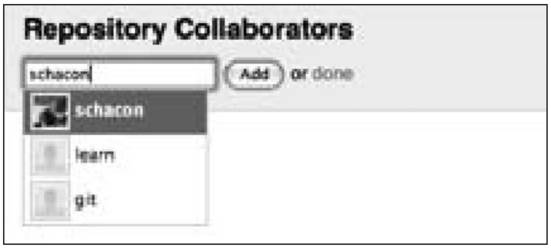
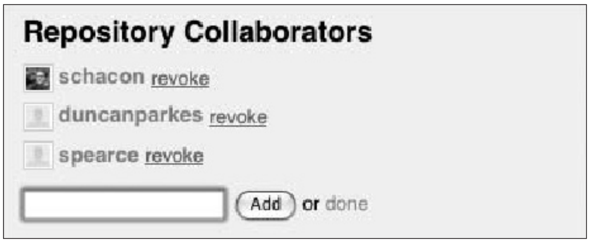


Figure 4-11. *Adding a collaborator to your project*

When you're finished adding collaborators, you should see a list of them in the Repository Collaborators box (see Figure 4-12).

If you need to revoke access to individuals, you can click the "revoke" link, and their push access will be removed. For future projects, you can also copy collaborator groups by copying the permissions of an existing project.



Repository Collaborators

schacon revoke

duncanparkes revoke

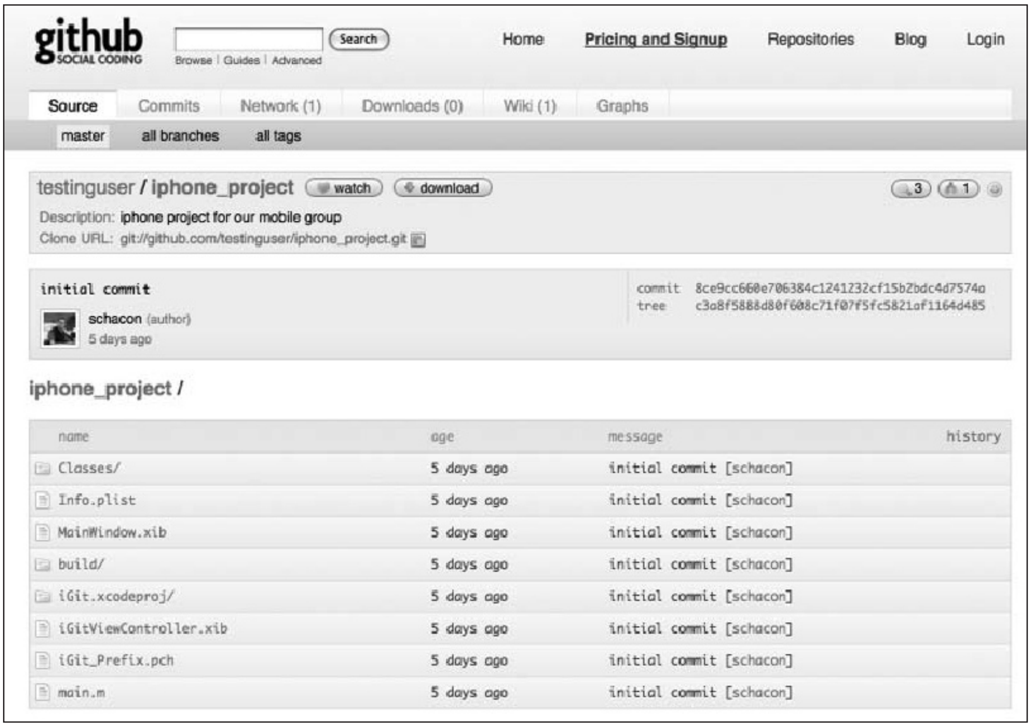
spearce revoke

Add or done

Figure 4-12. *A list of collaborators on your project*

Your Project

After you push your project up or have it imported from Subversion, you have a main project page that looks something like Figure 4-13.



testinguser / iphone_project

Description: iphone project for our mobile group

Clone URL: git://github.com/testinguser/iphone_project.git

```
initial commit
schacon (author)
5 days ago

commit  8ce9cc660e706384c1241232cf15b2bdc4d7574a
tree    c3a8f5888d80f608c71f07f5fc5821af1164d485
```

iphone_project /

name	age	message	history
Classes/	5 days ago	initial commit [schacon]	
Info.plist	5 days ago	initial commit [schacon]	
MainWindow.xib	5 days ago	initial commit [schacon]	
build/	5 days ago	initial commit [schacon]	
iGit.xcodeproj/	5 days ago	initial commit [schacon]	
iGitViewController.xib	5 days ago	initial commit [schacon]	
iGit_Prefix.pch	5 days ago	initial commit [schacon]	
main.m	5 days ago	initial commit [schacon]	

Figure 4-13. *A GitHub main project page*

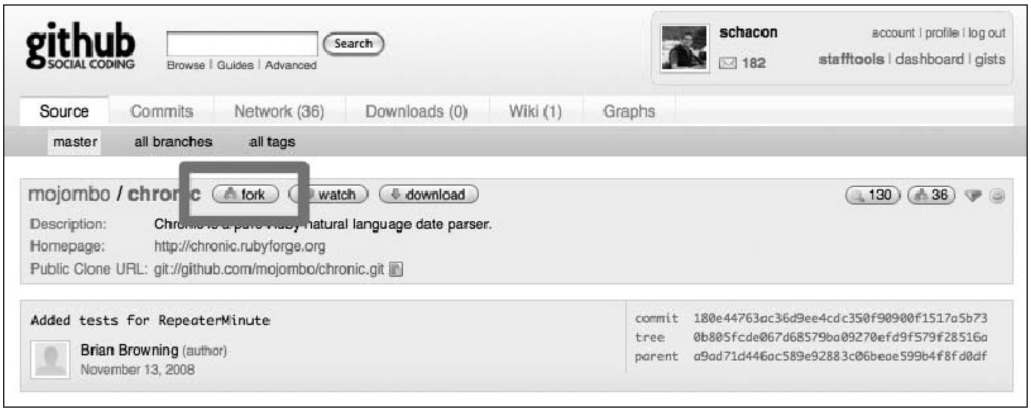
When people visit your project, they see this page. It contains tabs to different aspects of your projects. The Commits tab shows a list of commits in reverse chronological order, similar to the output of the `git log` command. The Network tab shows all the people who have forked your project and contributed back. The Downloads tab allows you to upload project binaries and link to tarballs and zipped versions of any tagged points in your project. The Wiki tab provides a wiki where you can write documentation or other information about your project. The Graphs tab has some contribution visualizations and statistics about your project. The main Source tab that you land on shows your project's main directory listing and automatically renders the `README` file below it if you have one. This tab also shows a box with the latest commit information.

Forking Projects

If you want to contribute to an existing project to which you don't have push access, GitHub encourages forking the project. When you land on a project page that looks interesting and you want to hack on it a bit, you can click the "fork" button in the project header to have GitHub copy that project to your user so you can push to it.

This way, projects don't have to worry about adding users as collaborators to give them push access. People can fork a project and push to it, and the main project maintainer can pull in those changes by adding them as remotes and merging in their work.

To fork a project, visit the project page (in this case, `mojombo/chronic`) and click the "fork" button in the header (see Figure 4-14).



Description: Chr[illegible] natural language date parser.
Homepage: http://chronic.rubyforge.org
Public Clone URL: git://github.com/mojombo/chronic.git

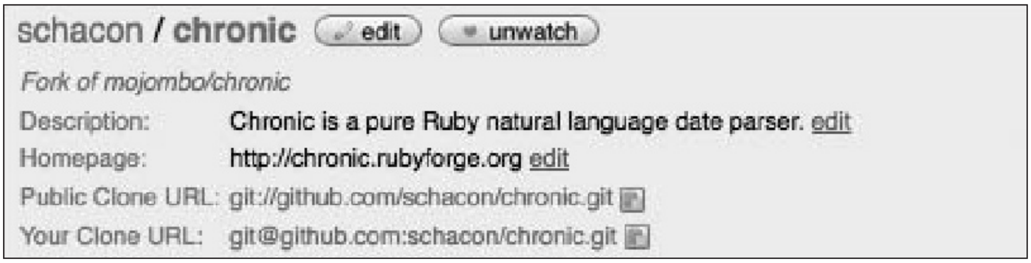
```
Added tests for RepeaterMinute
```

Brian Browning (author)
November 13, 2008

```
commit  180e44763ac36d9ee4cdc350f90900f1517a5b73
tree    0b805fcde067d68579ba09270efd9f579f28516a
parent  a9ad71d446ac589e92883c06beae599b4f8fd0df
```

Figure 4-14. *Get a writable copy of any repository by clicking the "fork" button.*

After a few seconds, you're taken to your new project page, which indicates that this project is a fork of another one (see Figure 4-15).



Fork of mojombo/chronic

Description:	Chronic is a pure Ruby natural language date parser. edit
Homepage:	http://chronic.rubyforge.org edit
Public Clone URL:	git://github.com/schacon/chronic.git
Your Clone URL:	git@github.com:schacon/chronic.git

Figure 4-15. *Your fork of a project*

GitHub Summary

That's all I'll cover about GitHub, but it's important to note how quickly you can do all this. You can create an account, add a new project, and push to it in a matter of minutes. If your project is open source, you also get a huge community of developers who now have visibility into your project and may well fork it and help contribute to it. At the very least, this may be a way to get up and running with Git and try it out quickly.

Summary

You have several options to get a remote Git repository up and running so that you can collaborate with others or share your work.

Running your own server gives you a lot of control and allows you to run the server within your own firewall, but such a server generally requires a fair amount of your time to set up and maintain. If you place your data on a hosted server, it's easy to set up and maintain; however, you have to be able to keep your code on someone else's servers, and some organizations don't allow that.

It should be fairly straightforward to determine which solution or combination of solutions is appropriate for you and your organization.