



Simplify Your Life With an SSH Config File

929 words — written on March 17th, 2011 by [Joël Perras](#).

If you're anything like me, you probably log in and out of a half dozen remote servers (or these days, local virtual machines) on a daily basis. And if you're even *more* like me, you have trouble remembering all of the various usernames, remote addresses and command line options for things like specifying a non-standard connection port or forwarding local ports to the remote machine.

Shell Aliases

Let's say that you have a remote server named `dev.example.com`, which has *not* been set up with public/private keys for password-less logins. The username to the remote account is *fooey*, and to reduce the number of scripted login attempts, you've decided to change the default SSH port to `2200` from the normal default of `22`. This means that a typical command would look like:

```
$ ssh fooey@dev.example.com -p 22000  
password: *****
```

Not too bad.

We can make things simpler and more secure by using a public/private key pair; I highly recommend using [ssh-copy-id](#) for moving your public keys around. It will save you quite a few folder/file permission headaches.

```
$ ssh fooey@dev.example.com -p 22000
# Assuming your keys are properly setup...
```

Now this doesn't seem all that bad. To cut down on the verbosity you could create a simple alias in your shell as well:

```
$ alias dev='ssh fooey@dev.example.com -p 22000'
$ dev # To connect
```

This works surprisingly well: Every new server you need to connect to, just add an alias to your `.bashrc` (or `.zshrc` if you hang with the cool kids), and voilà.

~/.ssh/config

However, there's a much more elegant and flexible solution to this problem. Enter the SSH config file:

```
# contents of $HOME/.ssh/config
Host dev
    HostName dev.example.com
    Port 22000
    User fooey
```

This means that I can simply `$ ssh dev`, and the options will be read from the configuration file. Easy peasy. Let's see what else we can do with just a few simple configuration directives.

Personally, I use quite a few public/private keypairs for the various servers and services that I use, to ensure that in the event of having one of my keys compromised the damage is as restricted as possible. For example, I have a key that I use uniquely for my [Github](#) account. Let's set it up so that that particular private key is used for all my github-related operations:

```
Host dev
```

```
HostName dev.example.com
Port 22000
User fooey</p>
Host github.com
  IdentityFile ~/.ssh/github.key
```

The use of `IdentityFile` allows me to specify exactly which private key I wish to use for authentication with the given host. You can, of course, simply specify this as a command line option for "normal" connections:

```
$ ssh -i ~/.ssh/blah.key username@host.com
```

but the use of a config file with `IdentityFile` is **pretty much your only option** if you want to specify which identity to use for any git commands. This also opens up the very interesting concept of further segmenting your github keys on something like a per-project or per-organization basis:

```
Host github-project1
  User git
  HostName github.com
  IdentityFile ~/.ssh/github.project1.key</p>
Host github-org
  User git
  HostName github.com
  IdentityFile ~/.ssh/github.org.key</p>
Host github.com
  User git
  IdentityFile ~/.ssh/github.key
```

Which means that if I want to clone a repository using my organization credentials, I would use the following:

```
$ git clone git@github-org:orgname/some_repository.git
```

Going further

As any security-conscious developer would do, I set up

firewalls on all of my servers and make them as restrictive as possible; in many cases, this means that the only ports that I leave open are `80/443` (for webserver), and port `22` for SSH (or whatever I might have remapped it to for obfuscation purposes). On the surface, this seems to prevent me from using things like a desktop MySQL GUI client, which expect port `3306` to be open and accessible on the remote server in question. The informed reader will note, however, that a simple local port forward can save you:

```
$ ssh -f -N -L 9906:127.0.0.1:3306 coolio@database.example.com
# -f puts ssh in background
# -N makes it not execute a remote command
```

This will forward all local port `9906` traffic to port `3306` on the remote `dev.example.com` server, letting me point my desktop GUI to localhost (`127.0.0.1:9906`) and have it behave exactly as if I had exposed port `3306` on the remote server and connected directly to it.

Now I don't know about you, but remembering that sequence of flags and options for `SSH` can be a complete pain. Luckily, our config file can help alleviate that:

```
Host tunnel
  HostName database.example.com
  IdentityFile ~/.ssh/coolio.example.key
  LocalForward 9906 127.0.0.1:3306
  User coolio
```

Which means I can simply do:

```
$ ssh -f -N tunnel
```

And my local port forwarding will be enabled using all of the configuration directives I set up for the tunnel host. Slick.

Homework

There are quite a few configuration options that you can specify in `~/.ssh/config`, and I highly suggest consulting the online [documentation](#) or the **ssh_config** man page. Some interesting/useful things that you can do include: change the default number of connection attempts, specify local environment variables to be passed to the remote server upon connection, and even the use of `*` and `?` wildcards for matching hosts.

I hope that some of this is useful to a few of you. Leave a note in the comments if you have any cool tricks for the SSH config file; I'm always on the lookout for fun hacks.

Related Posts

[Installing Matplotlib, NumPy, SciPy, NLTK, and Pandas on OS X Without Going Crazy](#) September 3rd, 2014

[Bartering for Beers with Approximate Subset Sums](#) August 19th, 2014

[Ideas of March](#) March 15th, 2011