Git Setup & configurations

Back

First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once; they'll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three

different places:

/etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option--system to git config, it reads and writes from this file specifically.
~/.gitconfig file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
config file in the git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository. Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.

On Windows systems, Git looks for the .gitconfig file in the $HOME directory (%USERPROFILE% in Windows' environment), which is C:\Documents and Settings\$USER or C:\Users\$USER for most people, depending on version ($USER is %USERNAME% in Windows' environment). It also still looks for /etc/gitconfig, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer.

Your Identity

The first thing you should do when you install Git is to set your user name and e-mail address. This is important because every Git commit uses this information, and it's immutably baked into the commits you pass around:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the --global option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or e-mail address for specific projects, you can run the command without the --global option when you're in that project.

Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. By default, Git uses your system's default editor, which is generally Vi or Vim. If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

Your Diff Tool

Another useful option you may want to configure is the default diff tool to use to resolve merge conflicts. Say you want to use vimdiff:

```
$ git config --global merge.tool vimdiff
```

Git accepts kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff as valid merge tools. You can also set up a custom tool; see Chapter 7 for more information about doing that.

Checking Your Settings

If you want to check your settings, you can use the git config --list command to list all the settings Git can find at that point:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different files (/etc/gitconfig and ~/.gitconfig, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key's value is by typing git config {key}:

```
$ git config user.name
Scott Chacon
```

Basic Client Configuration

The configuration options recognized by Git fall into two categories: client side and server side. The majority of the options are client side—configuring your personal working preferences. Although tons of options are available, I'll only cover the few that either are commonly used or can significantly affect your workflow. Many options are useful only in edge cases that I won't go over here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ git config --help
```

The manual page for git config lists all the available options in quite a bit of detail.

core.editor

By default, Git uses whatever you've set as your default text editor or else falls back to the Vi editor to create and edit your commit and tag messages. To change that default to something else, you can use the

core.editor setting:

```
$ git config --global core.editor emacs
```

Now, no matter what is set as your default shell editor variable, Git will fire up Emacs to edit messages.

commit.template

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at $HOME/.gitmessage.txt that looks like this:

```
subject line

what happened

[ticket: X]
```

To tell Git to use it as the default message that appears in your editor when you run git commit, set the commit.template configuration value:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Then, your editor will open to something like this for your placeholder commit message when you commit:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

If you have a commit-message policy in place, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

core.pager

The core.pager setting determines what pager is used when Git pages output such as log and diff. You can set it to more or to your favorite pager (by default, it’s less), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long it is.

user.signingkey

If you're making signed annotated tags (as discussed in Chapter 2), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

```
$ git config --global user.signingkey <gpg-key-id>
```

Now, you can sign tags without having to specify your key every time with the git tag command:

```
$ git tag -s <tag-name>
```

core.excludesfile

You can put patterns in your project's .gitignore file to have Git not see them as untracked files or try to stage them when you run git add on them, as discussed in Chapter 2. However, if you want another file outside of your project to hold those values or have extra values, you can tell Git where that file is with the core.excludesfile setting. Simply set it to the path of a file that has content similar to what a .gitignore file would have.

help.autocorrect

This option is available only in Git 1.6.1 and later. If you mistype a command in Git, it shows you something like this:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
     commit
```

If you set help.autocorrect to 1, Git will automatically run the command if it has only one match under this scenario.

Colors in Git

Git can color its output to your terminal, which can help you visually parse the output quickly and easily. A number of options can help you set the coloring to your preference.

color.ui

Git automatically colors most of its output if you ask it to. You can get very specific about what you want colored and how; but to turn on all the default terminal coloring, set color.ui to true:

```
$ git config --global color.ui true
```

When that value is set, Git colors its output if the output goes to a terminal. Other possible settings are false, which never colors the output, and always, which sets colors all the time, even if you're redirecting Git commands to a file or piping them to another command.

You'll rarely want color.ui = always. In most scenarios, if you want color codes in your redirected output, you can instead pass a --color flag to the Git command to force it to use color codes. The color.ui = true setting is almost always what you'll want to use.

color.*

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to true, false, or always:

```
color.branch
color.diff
color.interactive
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: normal, black, red, green, yellow, blue, magenta, cyan, or white. If you want an attribute like bold in the previous example, you can choose from bold, dim, ul, blink, and reverse.

See the git config manpage for all the subsettings you can configure, if you want to do that.