

# 8.1 Customizing Git - Git Configuration

So far, we've covered the basics of how Git works and how to use it, and we've introduced a number of tools that Git provides to help you use it easily and efficiently. In this chapter, we'll see how you can make Git operate in a more customized fashion, by introducing several important configuration settings and the hooks system. With these tools, it's easy to get Git to work exactly the way you, your company, or your group needs it to.

## Git Configuration

As you briefly saw in [Getting Started](#), you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and email address:

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

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an `/etc/gitconfig` file, which contains values for every user on the system and all of their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. You can make Git read and write to this file by passing the `--global` option.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. These values are specific to that single repository.

Each of these "levels" (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

---

### NOTE

Git's configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It's generally easier to run the `git config` command, though.

---

## 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.

Many, *many* configuration options are supported, but a large fraction of them are only useful in certain edge cases. We'll only be covering the most common and most useful here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at <http://git-scm.com/docs/git-config.html>.

### **core.editor**

By default, Git uses whatever you've set as your default text editor (`$VISUAL` or `$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, 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 `~/.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 ~/.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 your team has a commit-message policy, 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**

This setting determines which 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 they are.

### **user.signingkey**

If you're making signed annotated tags (as discussed in [Signing Your Work](#)), 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 [Ignoring Files](#).

But sometimes you want to ignore certain files for all repositories that you work with. If your computer is running Mac OS X, you're probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about files that end with a `~`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/.gitignore_global` file with these contents:

```
*~  
.DS_Store
```

...and you run `git config --global core.excludesfile ~/.gitignore_global`, Git will never again bother you about those files.

### **help.autocorrect**

If you mistype a command, it shows you something like this:

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

```
Did you mean this?  
checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master  
WARNING: You called a Git command named 'chekcout', which does not exist.  
Continuing under the assumption that you meant 'checkout'  
in 0.1 seconds automatically...
```

Note that “0.1 seconds” business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

## Colors in Git

Git fully supports colored terminal output, which greatly aids in visually parsing command 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, but there’s a master switch if you don’t like this behavior. To turn off all Git’s colored terminal output, do this:

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

The default setting is `auto`, which colors output when it’s going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You’ll rarely want this; 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 default setting is almost always what you’ll want.

### `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` (underline), `blink`, and `reverse` (swap foreground and background).

## External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from <http://www.perforce.com/downloads/Perforce/>. Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a

number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:

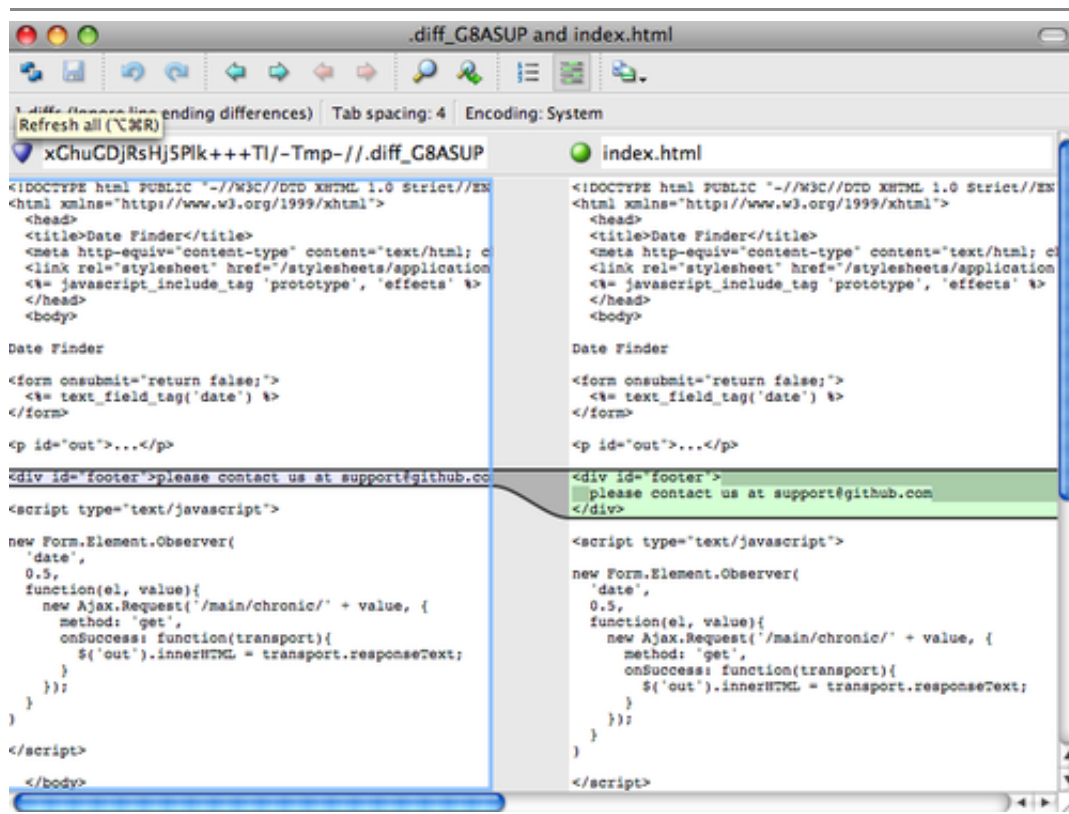


Figure 8-1. P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
  emerge
  gvimdiff
  gvimdiff2
  opendiff
  p4merge
  vimdiff
  vimdiff2
```

The following tools are valid, but not currently available:

```
araxis
bc3
codecompare
deltawalker
diffmerge
diffuse
ecmerge
kdiff3
meld
tkdiff
tortoisemerge
xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the `kdiff3` command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the `extMerge` and `extDiff` files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

## Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

### `core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:



```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to `input`:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

### **core.whitespace**

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The three that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

## Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

### `receive.fsckObjects`

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

### `receive.denyNonFastForwards`

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we'll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

### `receive.denyDeletes`

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn in [An Example Git-Enforced Policy](#).

