# 1.0 Getting Started - About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. We will use software source code as the files being version controlled via Git, though in reality you can do this with nearly any type of file on a computer.

# 1.1 A Short History of Git

The Linux kernel is an open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development

# 1.2 Git Basics

## Nearly Every Operation Is Local

Most operations in Git need only local files and resources to operate – generally no information is needed from another computer on your network.

For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you – it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

## Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

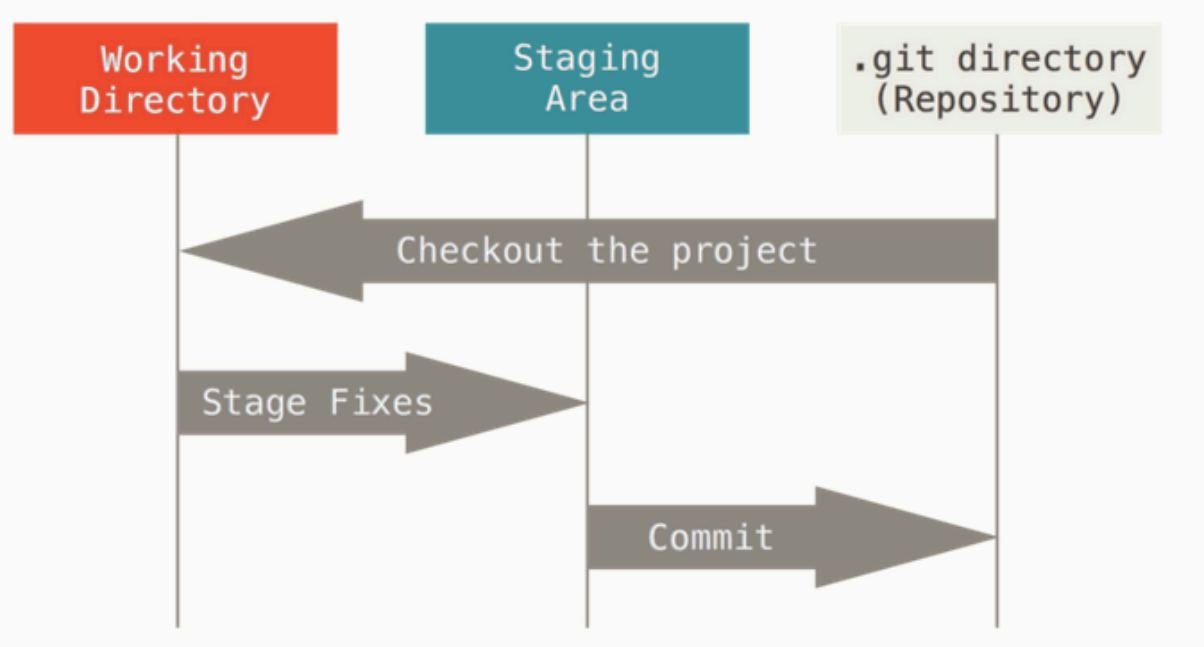
### Git Generally Only Adds Data

When you do actions in Git, nearly all of them only *add* data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

### The Three States

This is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working tree, and the staging area.




The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it's considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

### The Command Line

We will be using Git on the command line. For one, the command line is the only place you can run *all* Git commands – most of the GUIs implement only a partial subset of Git functionality for simplicity. If you know how to run the command-line version, you can probably also figure out how to run the GUI version, while the opposite is not necessarily true.

# 1.3 Installing Git

### Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora for example (or any closely-related RPM-based distro such as RHEL or CentOS), you can use `dnf`:

```
$ sudo dnf install git-all
```

If you're on a Debian-based distribution like Ubuntu, try `apt-get`:

```
$ sudo apt-get install git-all
```

For more options, there are instructions for installing on several different Unix flavors on the Git website, at http://git-scm.com/download/linux.

### Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to http://git-scm.com/download/win and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://git-for-windows.github.io/.

To get an automated installation you can use the Git Chocolatey package. Note that the Chocolatey package is community maintained.

Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub for Windows website, at http://windows.github.com.

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

# 1.4 First-Time Git Setup

## 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 on any given computer; 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:

1. `/etc/gitconfig` file: Contains values applied to 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.
2. `~/.gitconfig` or `~/.config/git/config` file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the `--global` option.
3. `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 (`C:\Users\$USER` for most people). 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. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in `C:\ProgramData\Git\config` on Windows Vista and newer. This config file can only be changed by `git config -f <file>` as an admin.

## Your Identity

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

```
$ 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 email address for specific projects, you can run the command without the `--global` option when you're in that project.

Many of the GUI tools will help you do this when you first run them.

### 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. If not configured, Git uses your system's default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

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

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

In the case of Notepad++, a popular programming editor, you are likely to want to use the 32-bit version, since at the time of writing the 64-bit version doesn't support all plug-ins. If you are on a 32-bit Windows system, or you have a 64-bit editor on a 64-bit system, you'll type something like this:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -
multiInst -nosession"
```

If you have a 32-bit editor on a 64-bit system, the program will be installed in `C:\Program Files (x86)`:

```
$ git config --global core.editor "'C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

### Checking Your Settings

If you want to check your configuration 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=John Doe
user.email=johndoe@example.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
John Doe
```

### Getting Help

If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example, you can get the manpage help for the `git config` command by running

```
$ git help config
```

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren't enough and you need in-person help, you can try the `#git` or `#github` channel on the Freenode IRC server (irc.freenode.net). These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

# 2.1 Getting a Git Repository

## Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

### Initializing a Repository in an Existing Directory

If you have a project that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory. If you've never done this, it looks a little different depending on which system you're running:

for Linux:

```
$ cd /home/user/my_project
```

for Mac:

```
$ cd /Users/user/my_project
```

for Windows:

```
$ cd /c/user/my_project
```

and type:

```
$ git init
```

This creates a new subdirectory named `.git` that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few `git add` commands that specify the files you want to track, followed by a `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

### Cloning an Existing Repository

If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is `git clone`. Git receives a full copy of nearly all data that the server has.

Every version of every file for the history of the project is pulled down by default when you run `git clone`.

You clone a repository with `git clone <url>`. For example, if you want to clone the Git linkable library called `libgit2`, you can do so like this: /* SHARE URL TO CLONE */

```
$ git clone https://github.com/libgit2/libgit2
```

That creates a directory named `libgit2`, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new `libgit2` directory, you'll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than `libgit2`, you can specify that as the next command-line option:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

That command does the same thing as the previous one, but the target directory is called `mylibgit`.

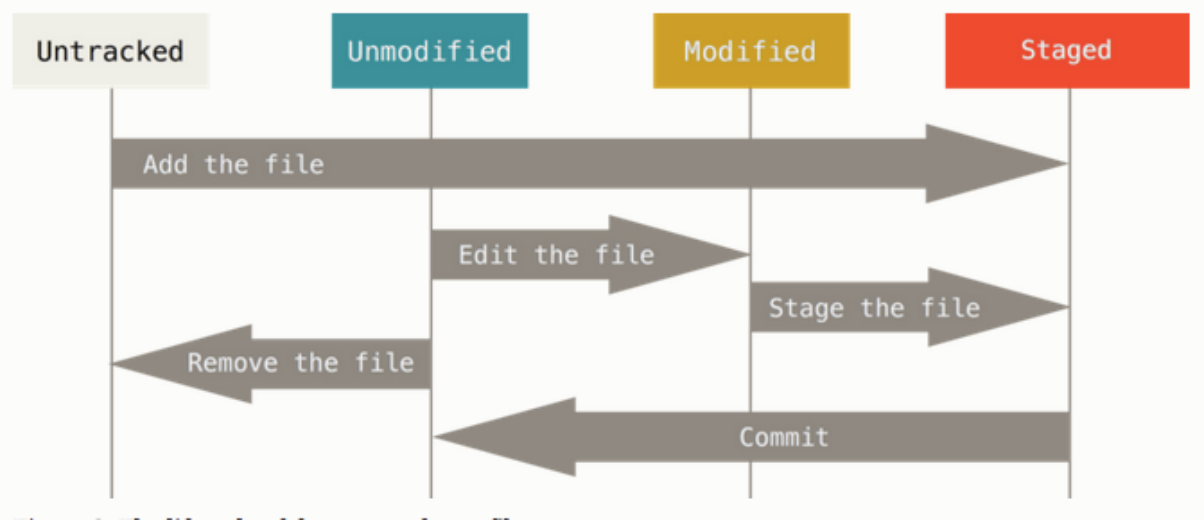
Git has a number of different transfer protocols you can use. The previous example uses the `https://` protocol, but you may also see `git://` or `user@server:path/to/repo.git`, which uses the SSH transfer protocol.

# 2.2 Recording Changes to the Repository

You have a bona fide Git repository and a checkout or working copy of the files for that project. You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.



## Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This means you have a clean working directory – in other words, none of your tracked files are modified.

Let's say you add a new file to your project, a simple `README` file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new `README` file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including `README`, so let's start tracking the file.

## Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the `README` file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your `README` file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add <files>` – that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

## Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called `CONTRIBUTING.md` and then run your `git status` command again, you get something that looks like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

The `CONTRIBUTING.md` file appears under a section named "Changes not staged for commit" – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as "add this content to the next commit" rather than "add this file to the project". Let's run `git add` now to stage the `CONTRIBUTING.md` file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you're ready to commit. However, let's run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

### Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on.

### Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`. Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

### Viewing Your Staged and Unstaged Changes

What have you changed but not yet staged? And what have you staged that you are about to commit?

Keep 1 file modified and 1 File staged and do a $ git status –

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

and `git diff --cached` to see what you've staged so far (`--staged` and `--cached` are synonyms)

## Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run `git add` on since you edited them – won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran `git status`, you saw that everything was staged, so you're ready to commit your changes. The simplest way to commit is to type `git commit`:

```
$ git commit
```

Doing so launches your editor of choice. (This is set by your shell's `EDITOR` environment variable – usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor`

Alternatively, you can type your commit message inline with the `commit` command by specifying it after a `-m` flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

### Skipping the Staging Area

```
$ git commit -a -m 'added new benchmarks'
```

### Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is, *unstaged*) area of your `git status` output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Then, if you run `git rm`, it stages the file's removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

The next time you commit, the file will be gone and no longer tracked.
If you modified the file and added it to the staging area already, you must force the removal with the `-f` option.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

```
$ git rm --cached README
```

You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as:

```
$ git rm log/\*.log
```

Note the backslash (`\`) in front of the `*`. This is necessary because Git does its own filename expansion in addition to your shell's filename expansion. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like this:

```
$ git rm \*~
```

This command removes all files whose names end with a `~`.

# 2.3 Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

```
$ git log

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

One of the more helpful options is `-p`, which shows the difference introduced in each commit. You can also use `-2`, which limits the output to only the last two entries:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
```

```
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
```

if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)
```

Also, try –

```
$ git log --pretty=oneline

$ git log --pretty=format:"%h - %an, %ar : %s"
```

And many more -
https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

## Limiting Log Output

```
$ git log --since=2.weeks
```

This command works with lots of formats – you can specify a specific date like `"2008-01-15"`, or a relative date such as `"2 years 1 day 3 minutes ago"`.

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages.

# 2.4 Undoing Things

/* TRY YOURSELVES */

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the `--amend` option:

```
$ git commit --amend
```

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit –amend
```

### Unstaging a Staged File

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Right below the “Changes to be committed” text, it says use `git reset HEAD <file>...` to unstage. So, let’s use that advice to unstage the `CONTRIBUTING.md` file:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M  CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

The command is a bit strange, but it works. The `CONTRIBUTING.md` file is modified but once again unstaged.

# 3.1 Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the internet or network. Either – Read Only or Read/Write

Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches.

```
$ git remote
origin

$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

### Adding Remote Repositories

```
$ git remote
origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

To get all the information that this repo has but you don't –

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Fetch from Remote –

```
$ git fetch <remote-name>
```

It's important to note that the `git fetch` command only downloads the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

You can use the `git pull` command to automatically fetch and then merge that remote branch into your current branch.

### Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push <remote-name> <branch-name>`.

```
$ git push origin master
#origin – Remote-name
#master – Branch-name
```

### Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show <remote-name>` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

### Removing and Renaming Remotes

```
# TO Rename
$ git remote rename pb paul
$ git remote
origin
paul
```

If you want to remove a remote for some reason – you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore – you can either use `git remote remove` or `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

# 3.2 Git Branching - Branches in a Nutshell

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

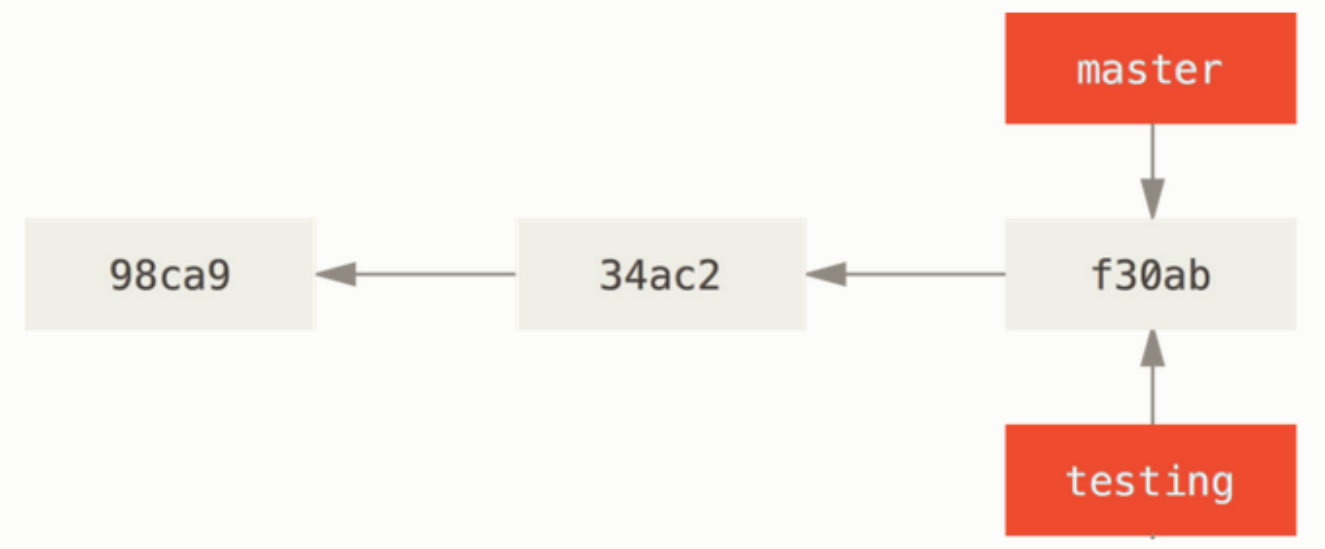
Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. **Incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast.**

Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

### Creating a New Branch

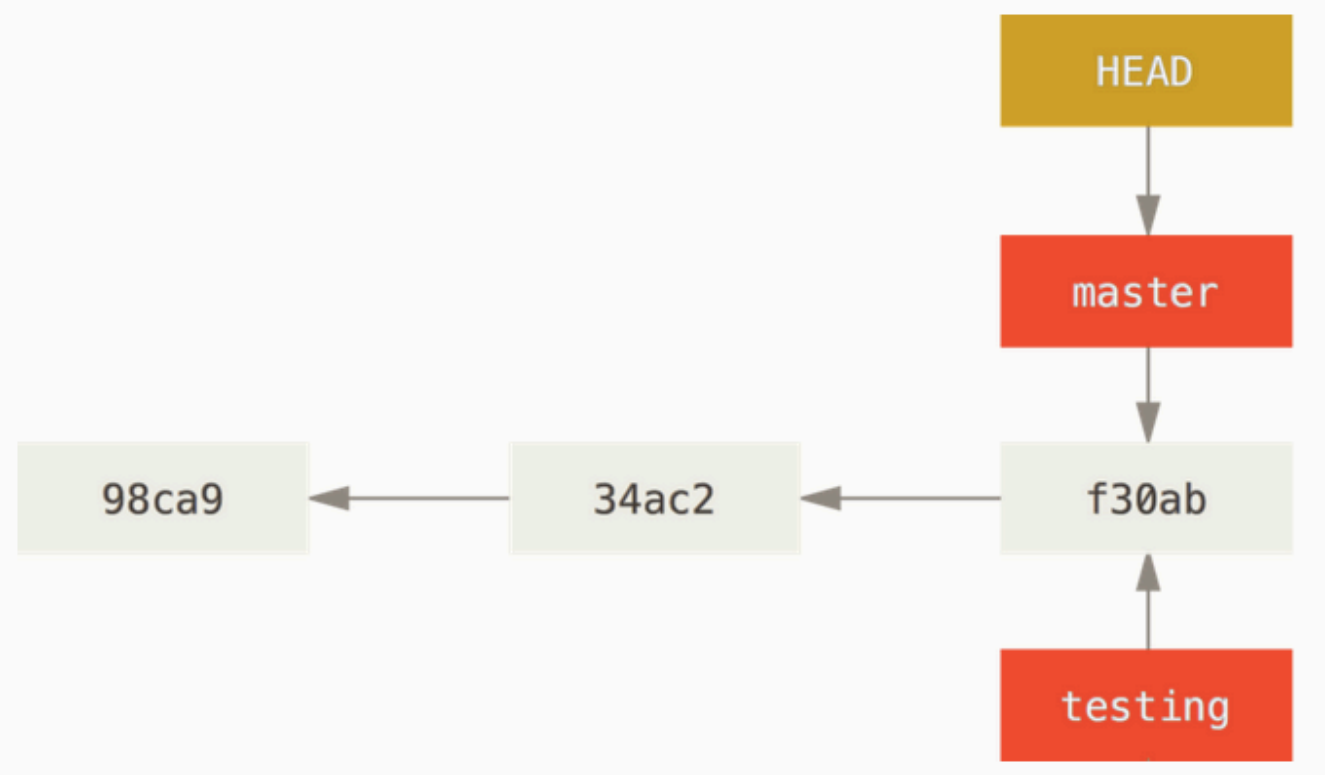
```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on.




Above, Two branches pointing into the same series of commits

How does Git know what branch you're currently on? It keeps a special pointer called `HEAD`.




You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to
the central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

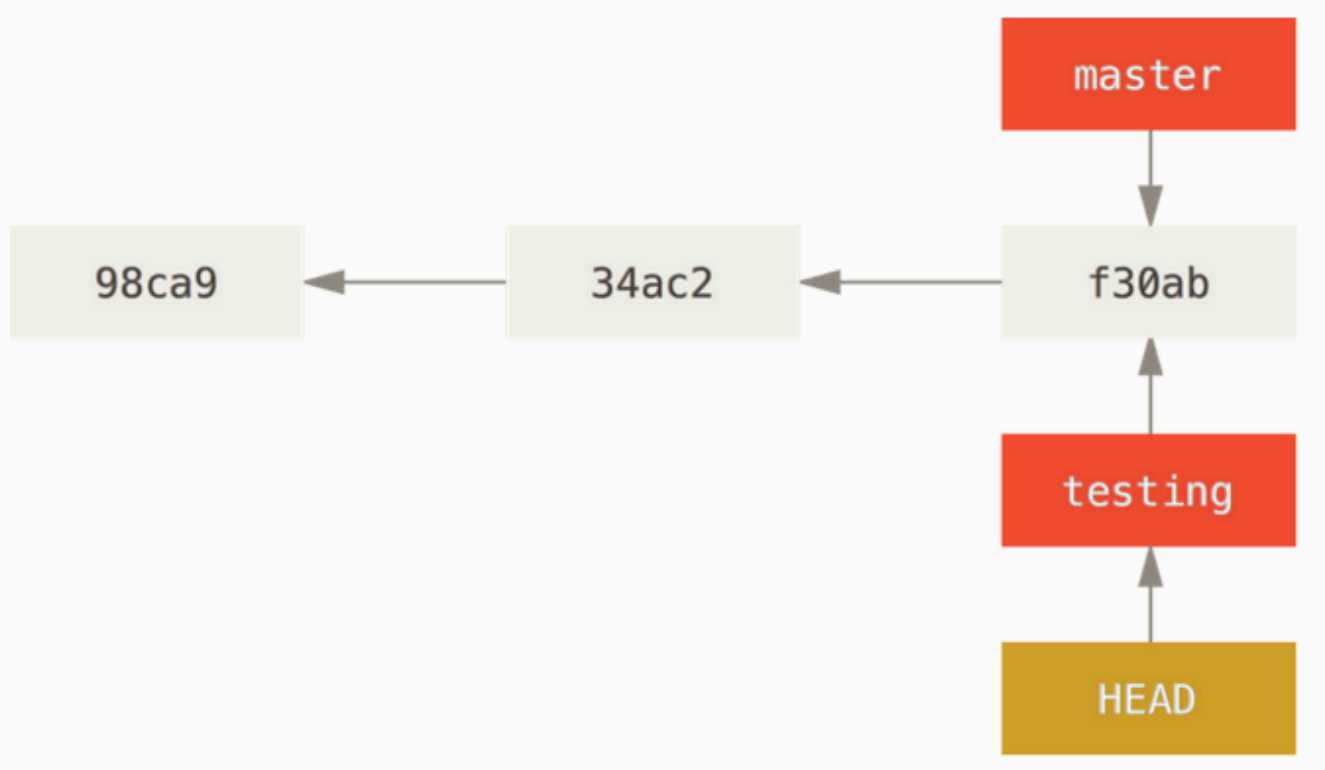
You can see the "master" and "testing" branches that are right there next to the `f30ab` commit.

## Switching Branches

To switch to an existing branch, you run the `git checkout` command. Let's switch to the new `testing` branch:

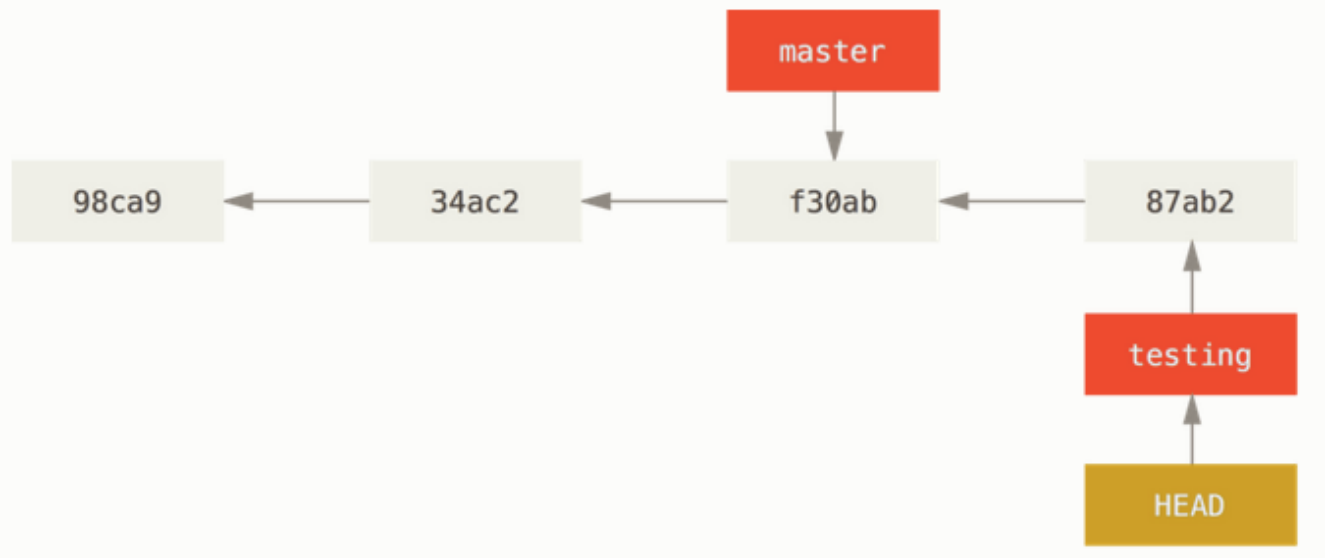
```
$ git checkout testing
```

This moves `HEAD` to point to the `testing` branch.



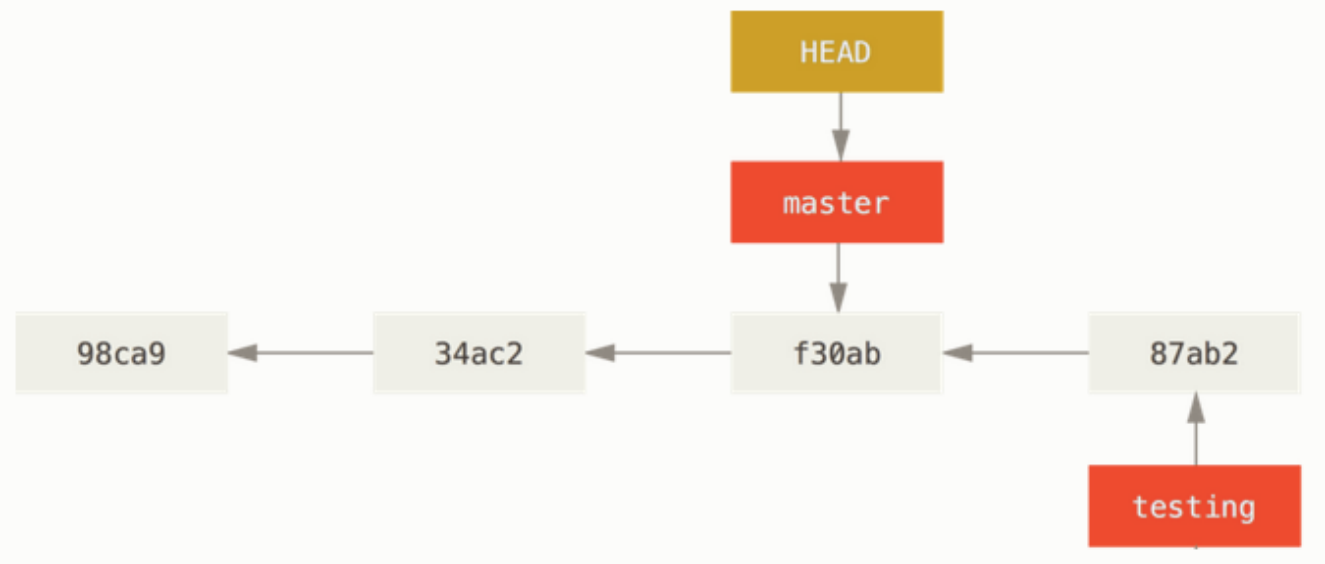

What is the significance of that? Well, let's do another commit:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



This is interesting, because now your `testing` branch has moved forward, but your `master` branch still points to the commit you were on when you ran `git checkout` to switch branches. Let's switch back to the `master` branch:

```
$ git checkout master
```




That command did two things: -

1. It moved the HEAD pointer back to point to the `master` branch, and
2. It reverted the files in your working directory back to the snapshot that `master` points to.

This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your `testing` branch so you can go in a different direction.

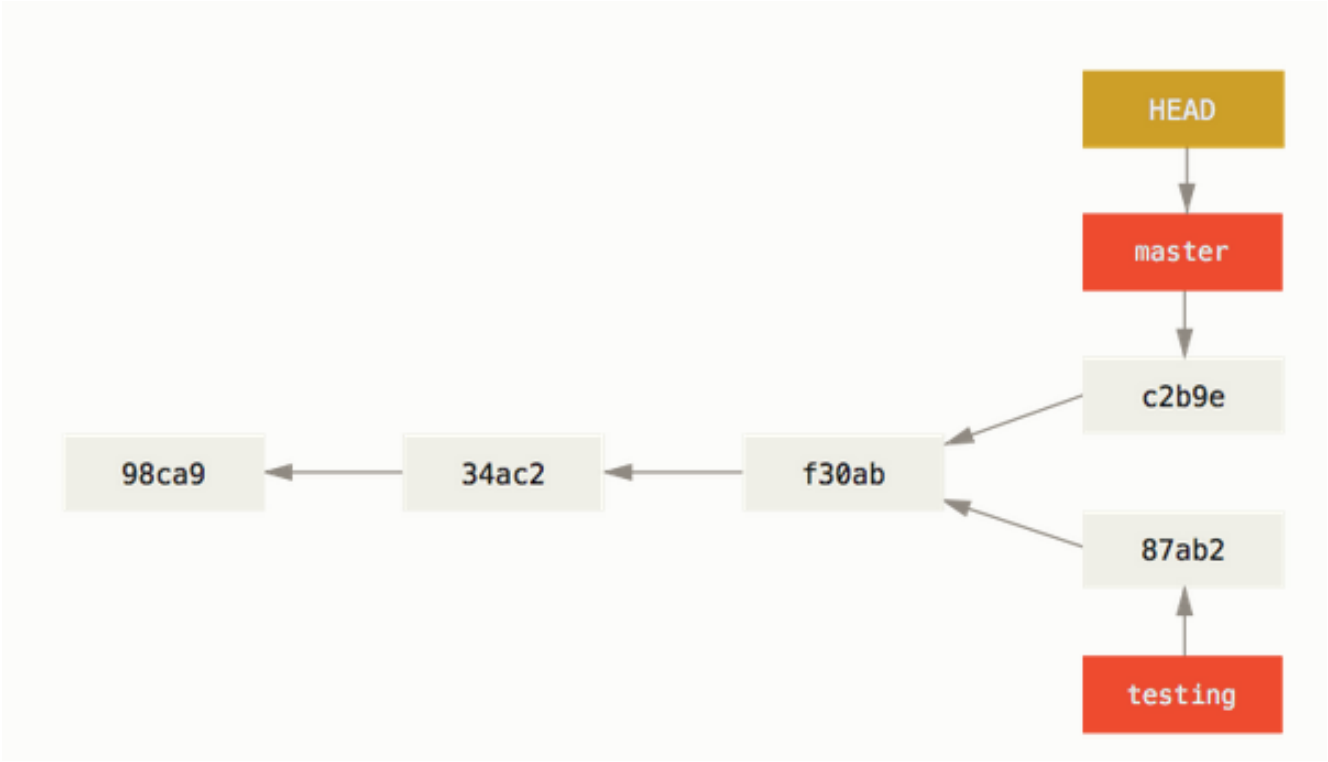
Let's make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Now your project history has diverged.

What you did ?
You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work.

What does it mean?
Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple `branch`, `checkout`, and `commit` commands.




You can also see this easily with the `git log` command. If you run `git log --oneline --decorate --graph --all` it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ git log --oneline --decorate --graph --all

* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Interesting Fact:
Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project's files into a second directory (which Git does not do).

# 3.3 Basic Branching and Merging

## Basic Branching and Merging

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

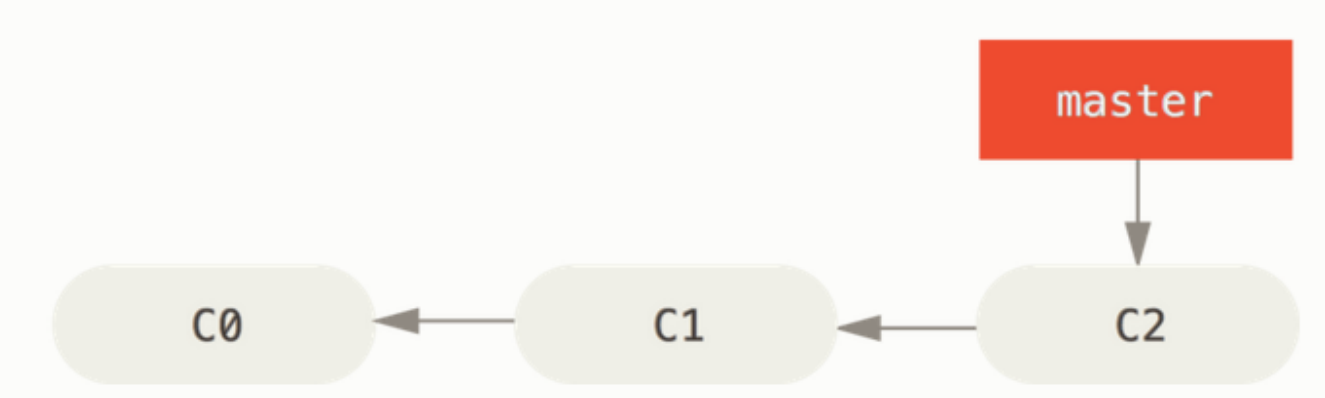
1. Do some work on a website.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

## Basic Branching

First, let's say you're working on your project and have a couple of commits already on the `master` branch.

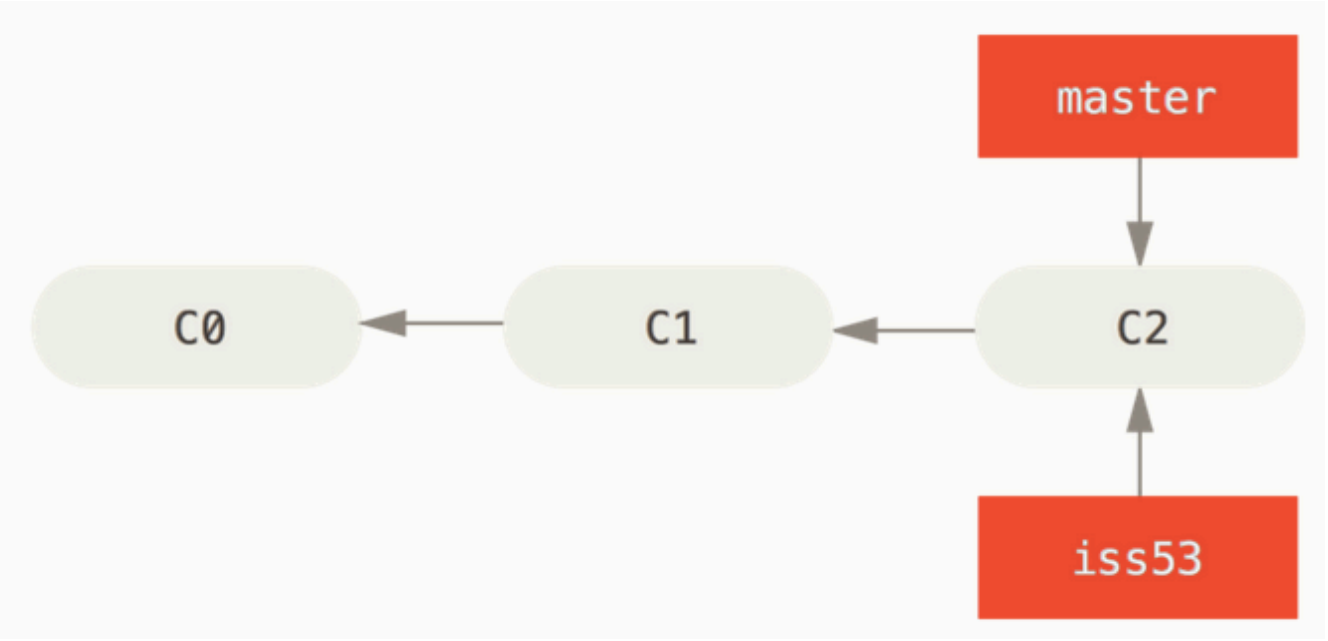



You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

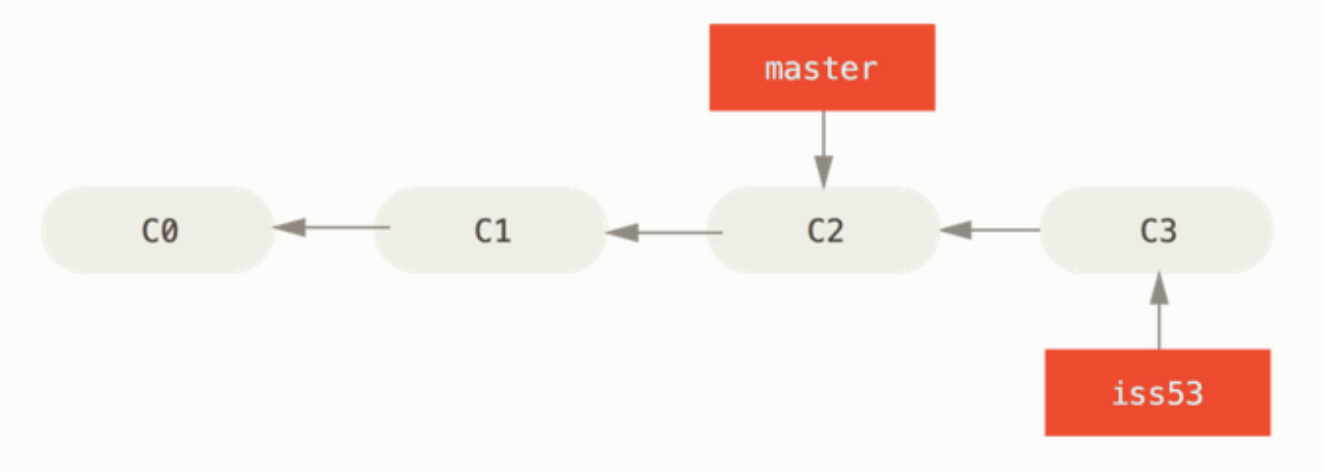
This is shorthand for:

```
$ git branch iss53
$ git checkout iss53
```




You work on your website and do some commits. Doing so moves the `iss53` branch forward, because you have it checked out (that is, your `HEAD` is pointing to it):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```




See - The `iss53` branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately.

All you have to do is switch back to your `master` branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches.

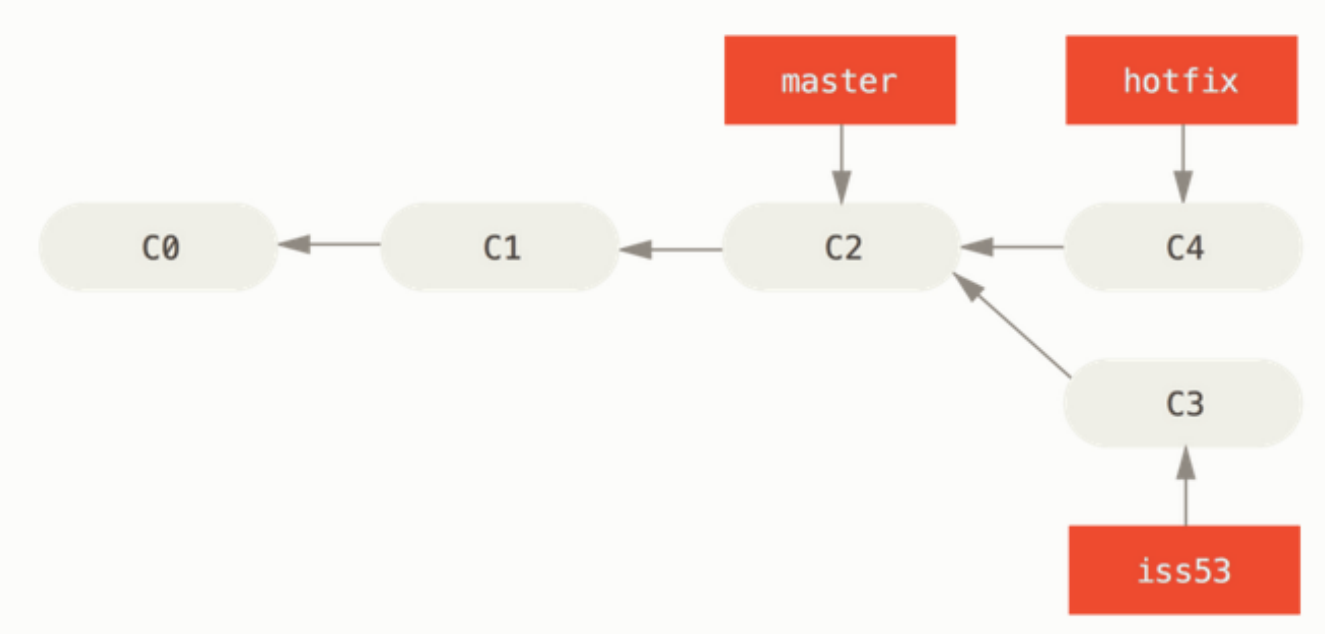
```
$ git checkout master
Switched to branch 'master'
```

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix.

Feature: -
This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a `hotfix` branch on which to work until it's completed:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

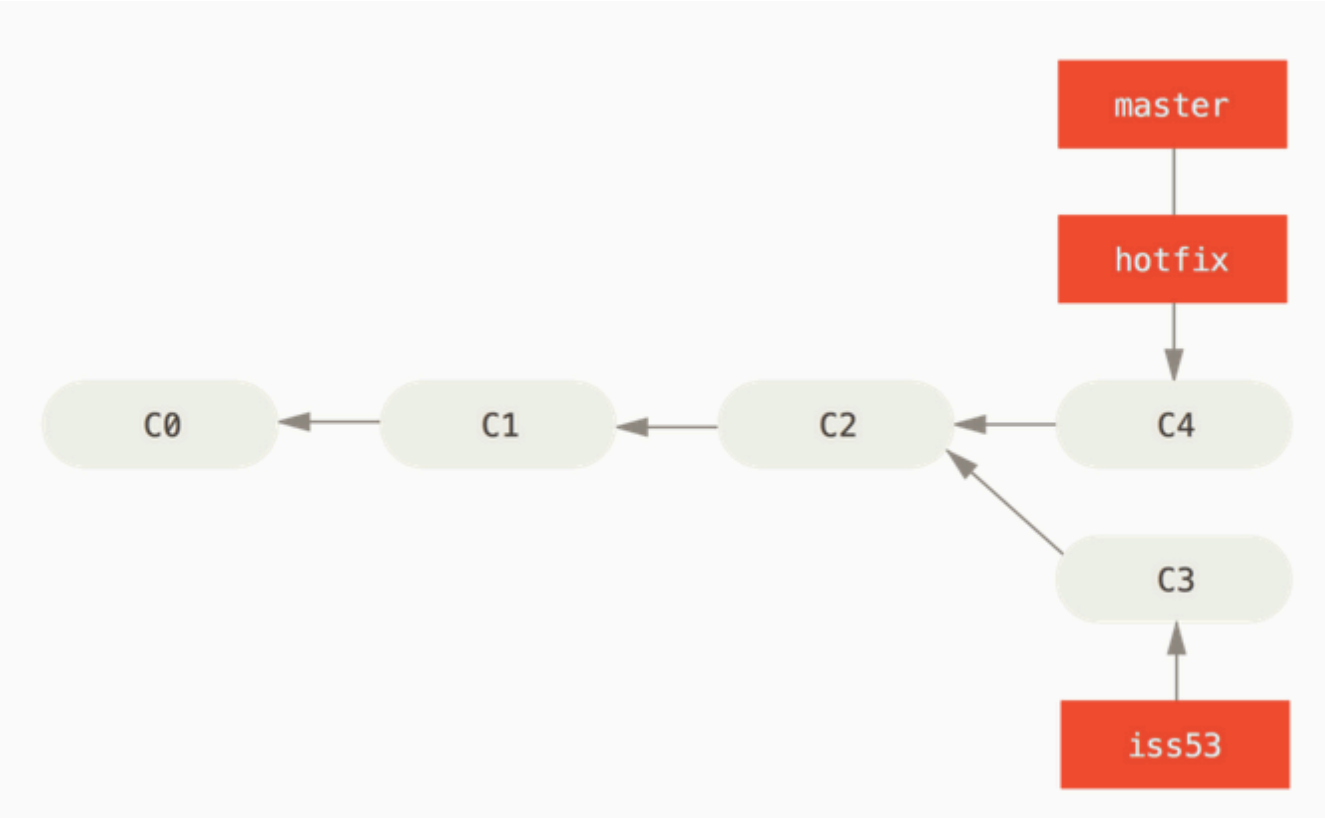



You can run your tests, make sure the hotfix is what you want, and finally merge the `hotfix` branch back into your `master` branch to deploy to production.

You do this with the `git merge` command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Why Fast forward? Discover More!
Because the commit `C4` pointed to by the branch `hotfix` you merged in was directly ahead of the commit `C2` you're on, Git simply moves the pointer forward.



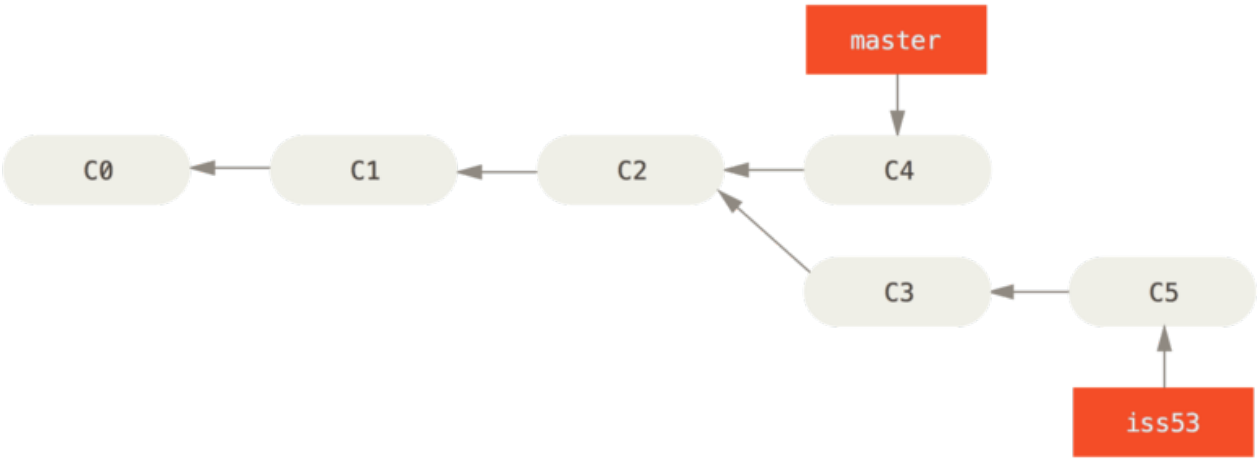

`master` is fast-forwarded to `hotfix`

After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted. However, first you'll delete the `hotfix` branch, because you no longer need it – the `master` branch points at the same place. You can delete it with the `-d` option to `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```




## Basic Merging

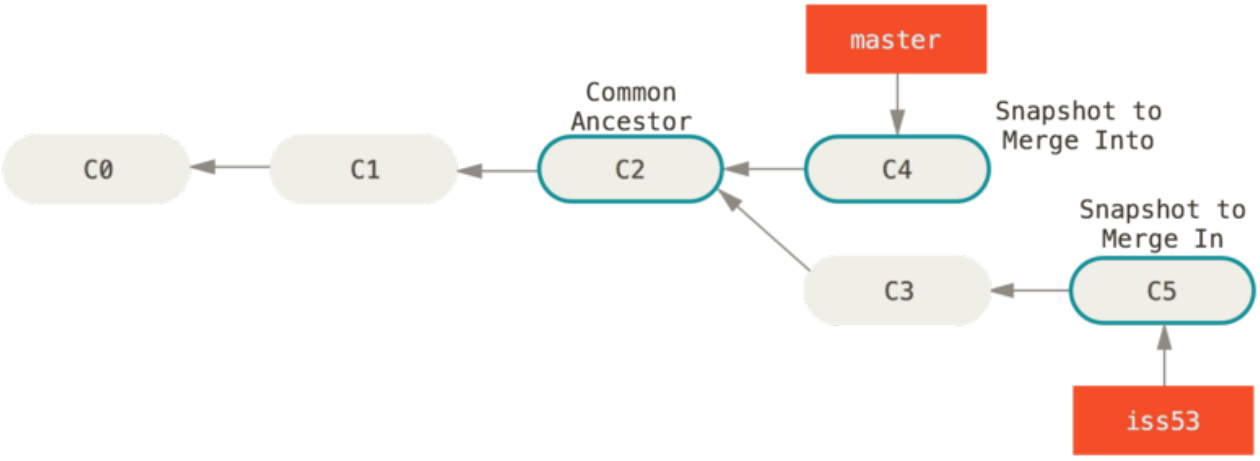
Suppose you've decided that your issue #53 work is complete and ready to be merged into your `master` branch.

Iss53 into the master branch –

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

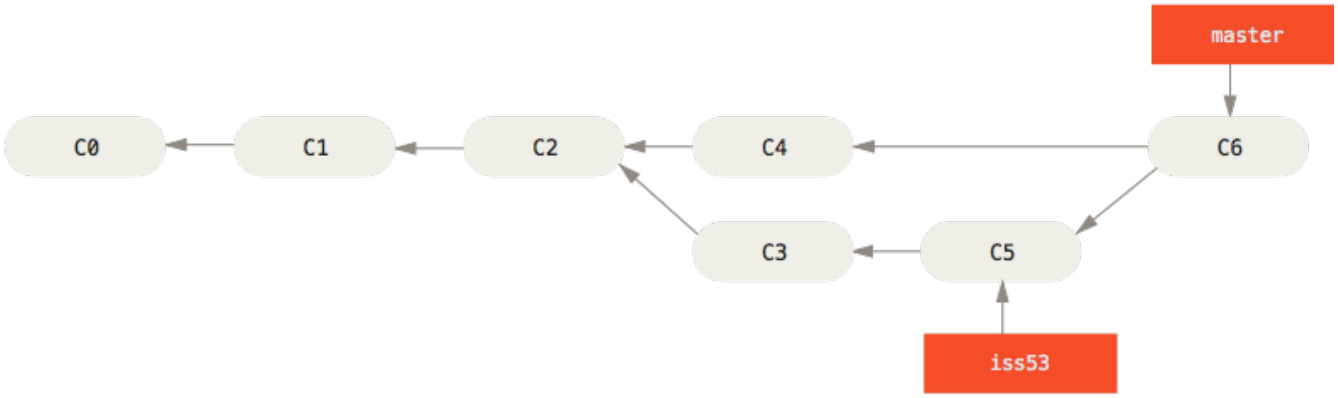
Notice – Recursive Strategy

This looks a bit different than the `hotfix` merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two. BEFORE MERGE : -



Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

AFTER MERGE : -




Now that your work is merged in, you have no further need for the `iss53` branch. You can close the branch –

```
$ git branch -d iss53
```

## Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the `hotfix` branch, you'll get a merge conflict that looks something like this:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
```

```
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

This means the version in `HEAD` (your `master` branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the `=======`), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution has a little of each section, and the `<<<<<<<`, `=======`, and `>>>>>>>` lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   index.html
#
```

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

# 3.4 Branch Management

The `git branch` command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
  iss53
* master
  testing
```

“ * ” indicates where the Branch Head points to.

To see the last commit on each branch, you can run `git branch -v`:

```
$ git branch -v
iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
testing 782fd34 add scott to the author list in the readmes
```

The useful `--merged` and `--no-merged` options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

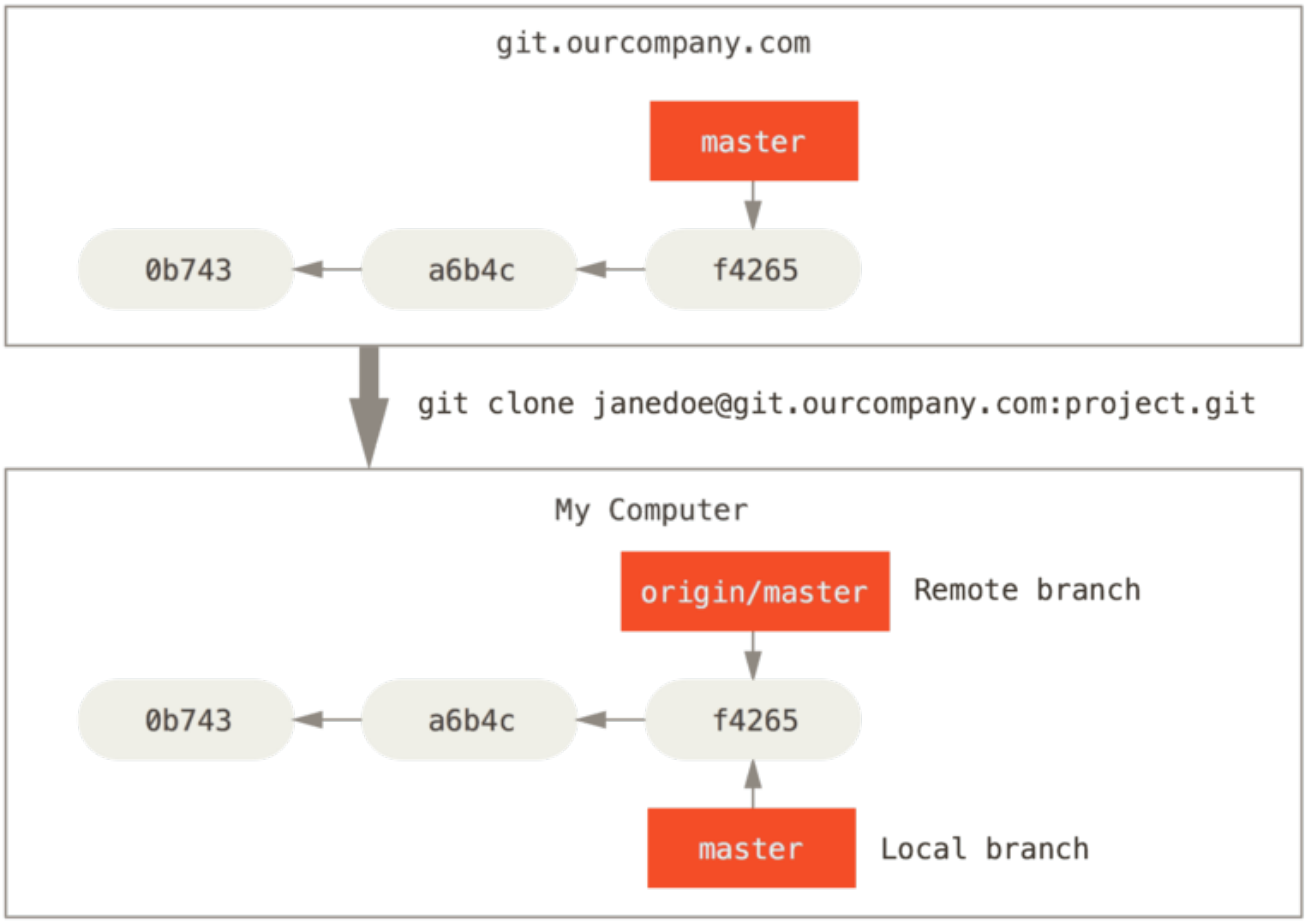
To see all the branches that contain work you haven’t yet merged in, you can run `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

This shows your other branch. Because it contains work that isn’t merged in yet, trying to delete it with `git branch -d` will fail:

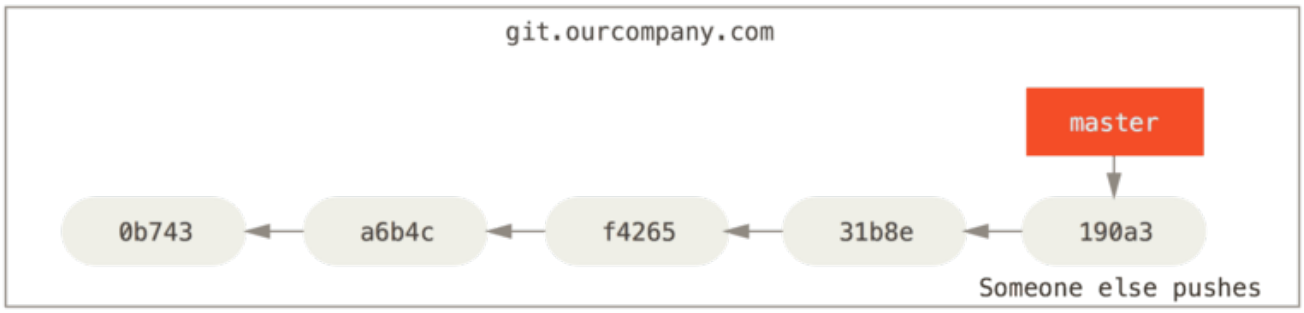
```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

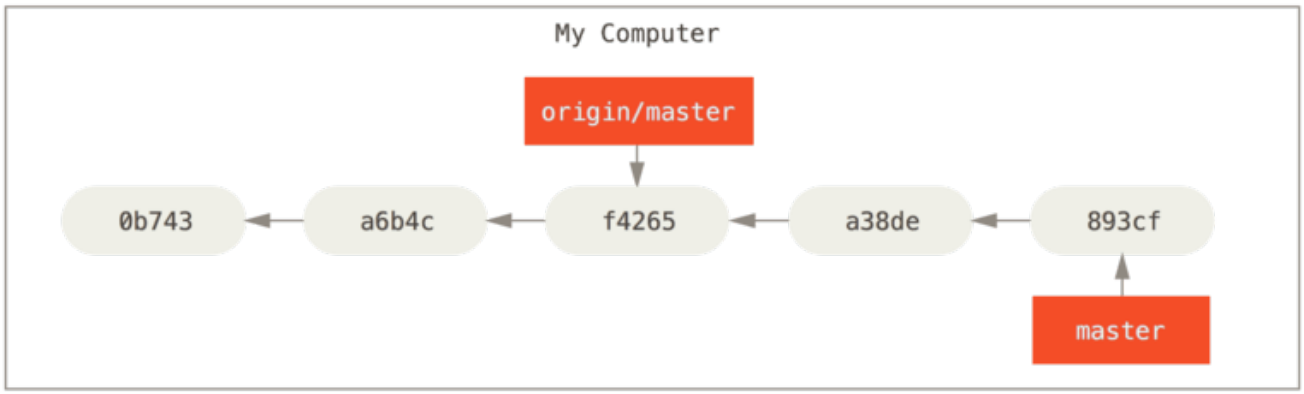
# 3.5 Remote Branches




Server and local repositories after cloning

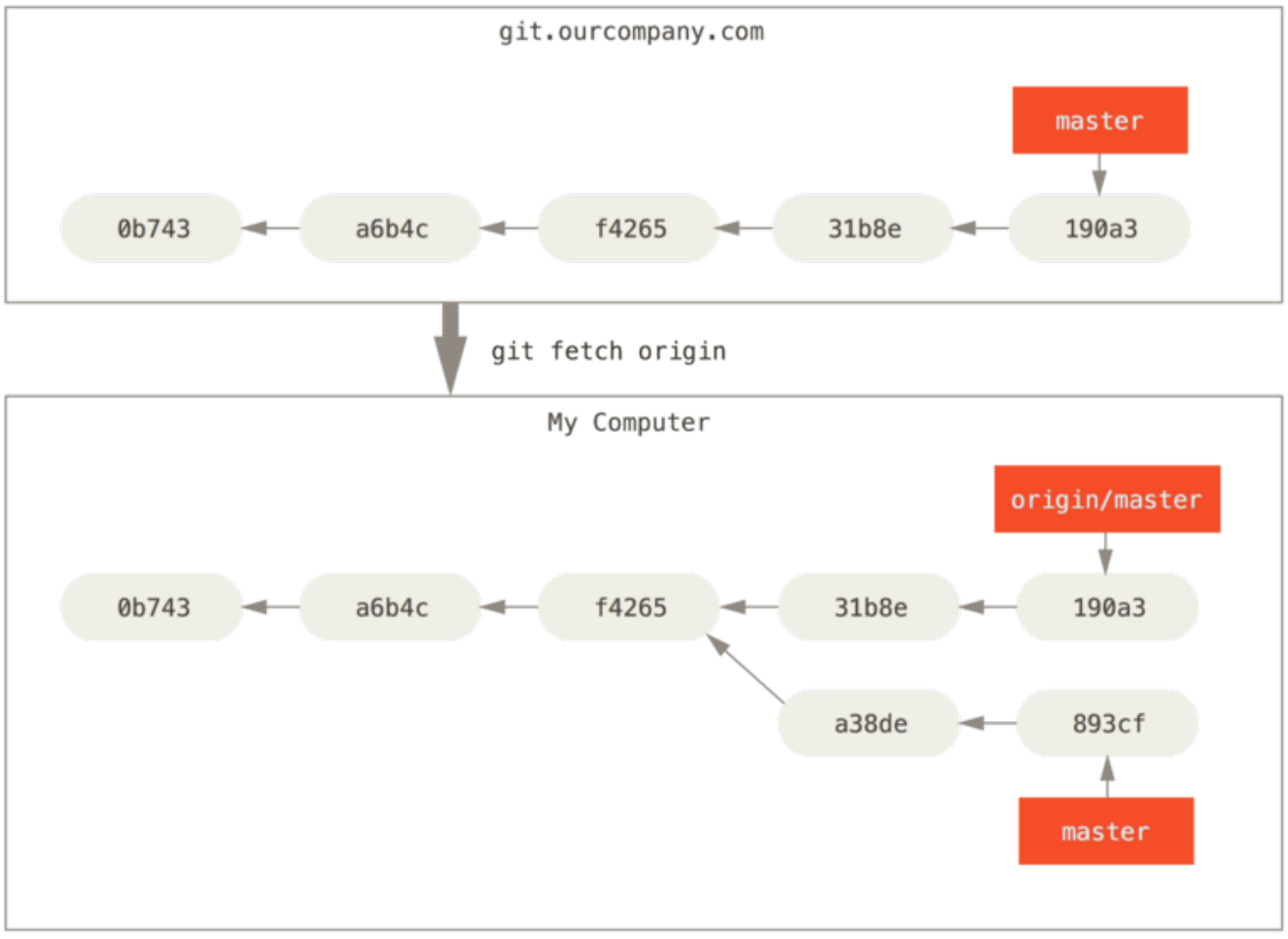
If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your `origin/master` pointer doesn't move.




Local and remote work can diverge

To synchronize your work, you run a `git fetch origin` command. This command looks up which server "origin" is (in this case, it's `git.ourcompany.com`), fetches any data from it that you don't yet have, and updates your local database, moving your `origin/master` pointer to its new, more up-to-date position.

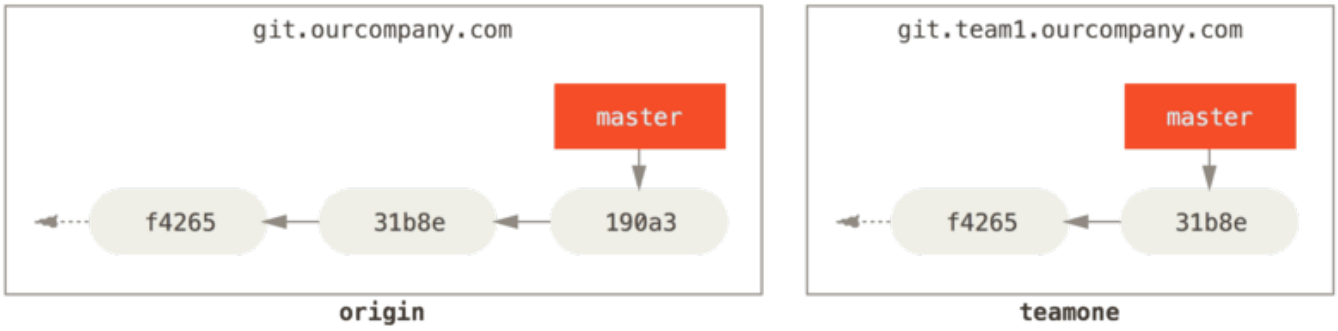


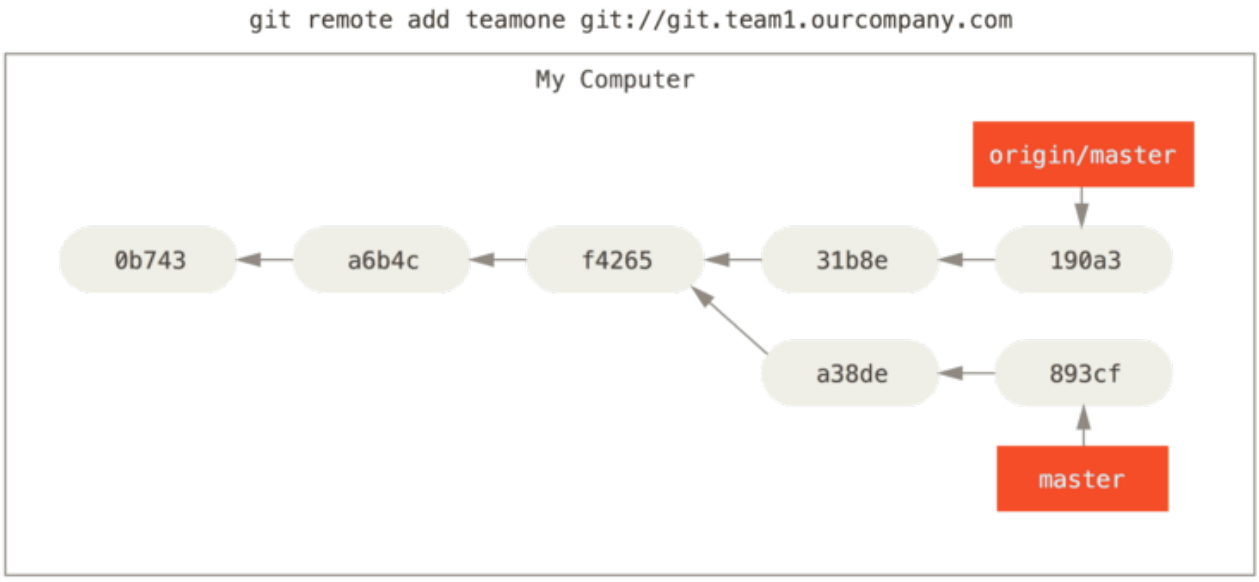
`git fetch` updates your remote references

MULTIPLE REMOTE SERVERS :

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`.

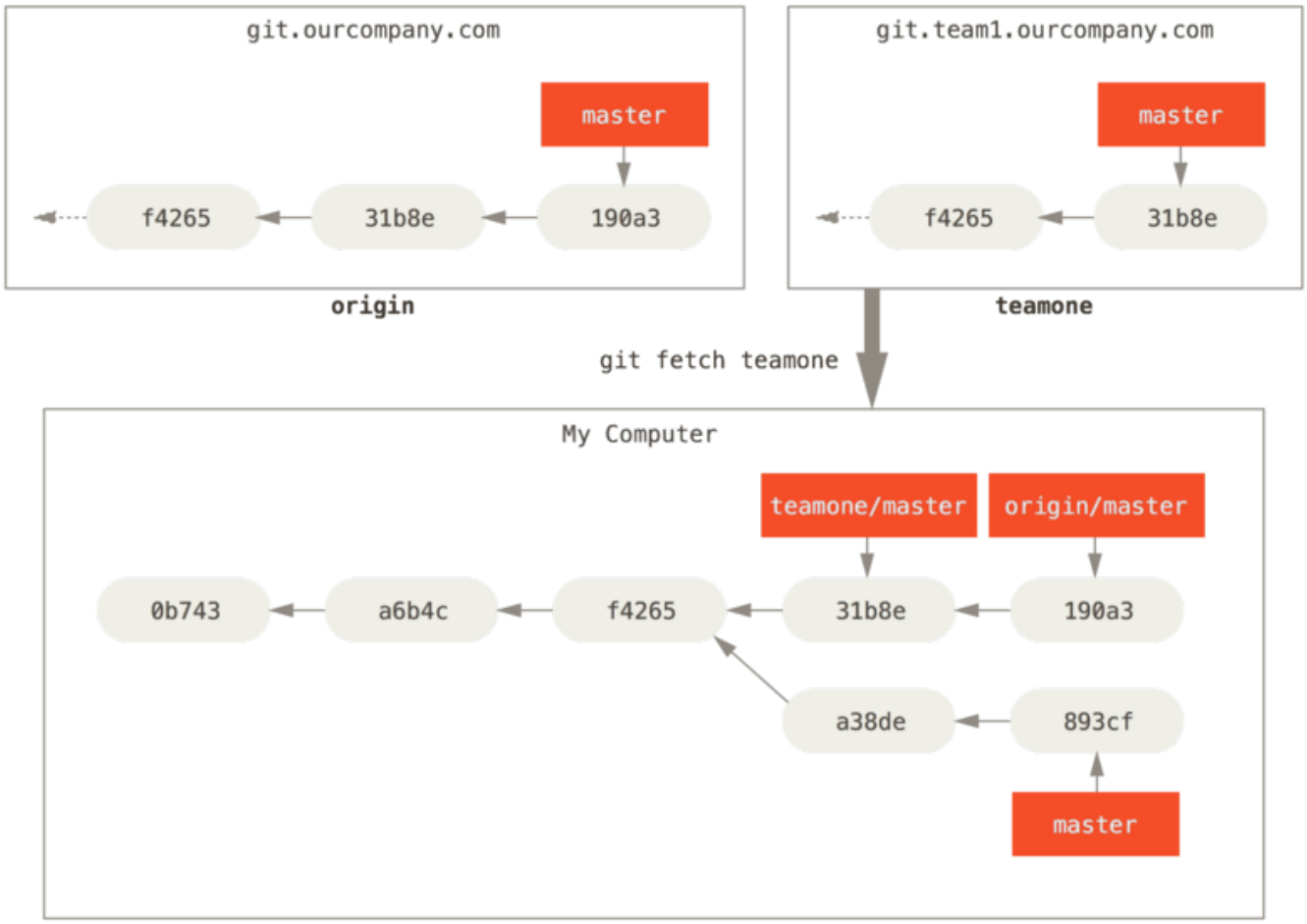
1. You can add it as a new remote reference to the project you're currently working on by running the `git remote add` command.
2. Name this remote `teamone`, which will be your shortname for that whole URL.




Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you don't have yet.

Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote-tracking branch called `teamone/master` to point to the commit that `teamone` has as its `master` branch.




## Pushing

When you want to share a branch with the world, you need to push it up to a remote that you have write access to.

Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share. That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

Run $ `git push <remote> <branch>`

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

## Tracking Branches – Advanced / Self Study!

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and branch to merge into.

When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`. However, you can set up other tracking branches if you wish – ones that track branches on other remotes, or don't track the `master` branch. The simple case is the example you just saw, running `git checkout -b <branch> <remotename>/<branch>`. This is a common enough operation that Git provides the `--track` shorthand:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

In fact, this is so common that there's even a shortcut for that shortcut. If the branch name you're trying to checkout (a) doesn't exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Now, your local branch `sf` will automatically pull from `origin/serverfix`.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you're tracking, you can use the `-u` or `--set-upstream-to` option to `git branch` to explicitly set it at any time.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

If you want to see what tracking branches you have set up, you can use the `-vv` option to `git branch`. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

So here we can see that our `iss53` branch is tracking `origin/iss53` and is "ahead" by two, meaning that we have two commits locally that are not pushed to the server. We can also see that our `master` branch is tracking `origin/master` and is up to date. Next we can see that our `serverfix` branch is tracking the `server-fix-good` branch on our `teamone` server and is ahead by three and behind by one, meaning that there is one commit on the server we haven't merged in yet and three commits locally that we haven't pushed. Finally we can see that our `testing` branch is not tracking any remote branch.

It's important to note that these numbers are only since the last time you fetched from each server. This command does not reach out to the servers, it's telling you about what it has cached from these servers locally. If you want totally up to date ahead and behind numbers, you'll need to fetch from all your remotes right before running this. You could do that like this:

```
$ git fetch --all; git branch -vv
```

### Pulling

While the `git fetch` command will fetch down all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, there is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases.

### Deleting Remote Branches

Suppose you're done with a remote branch – say you and your collaborators are finished with a feature and have merged it into your remote's `master` branch (or whatever branch your stable codeline is in). You can delete a remote branch using the `--delete` option to `git push`. If you want to delete your `serverfix` branch from the server, you run the following:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it's often easy to recover.

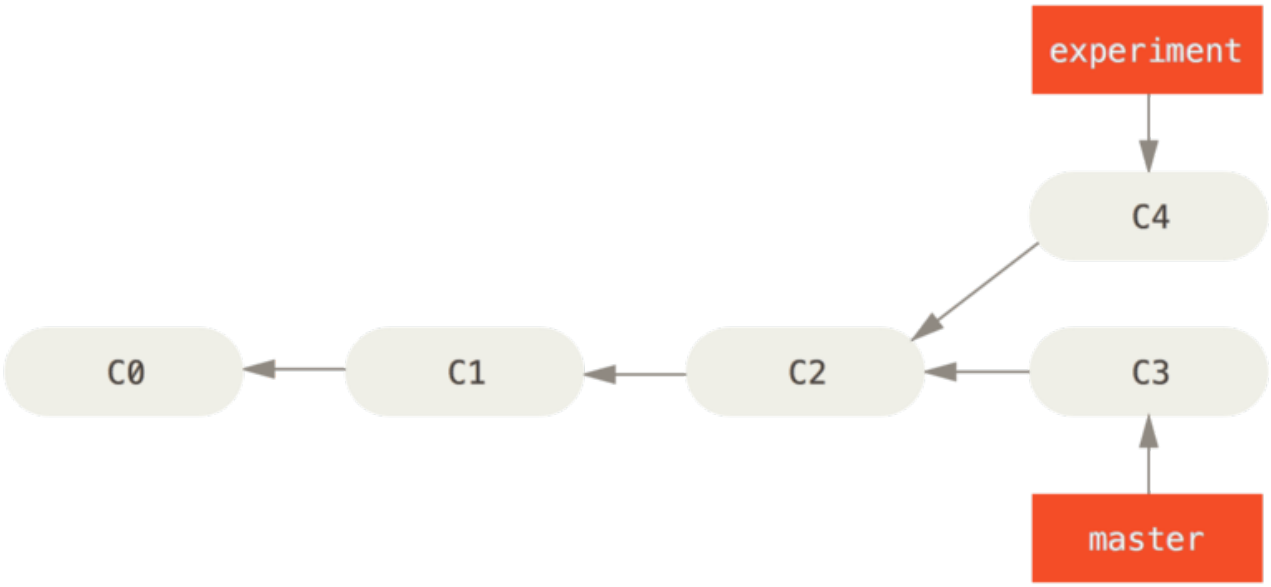
# 3.6 Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`.

Learn: How? Why? what cases you won't want to use it?

### The Basic Rebase

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.



The easiest way to integrate the branches, as we've already covered, is the `merge` command. It performs a three-way merge between the two latest branch snapshots (`C3` and `C4`) and the most recent common ancestor of the two (`C2`), creating a new snapshot (and commit).

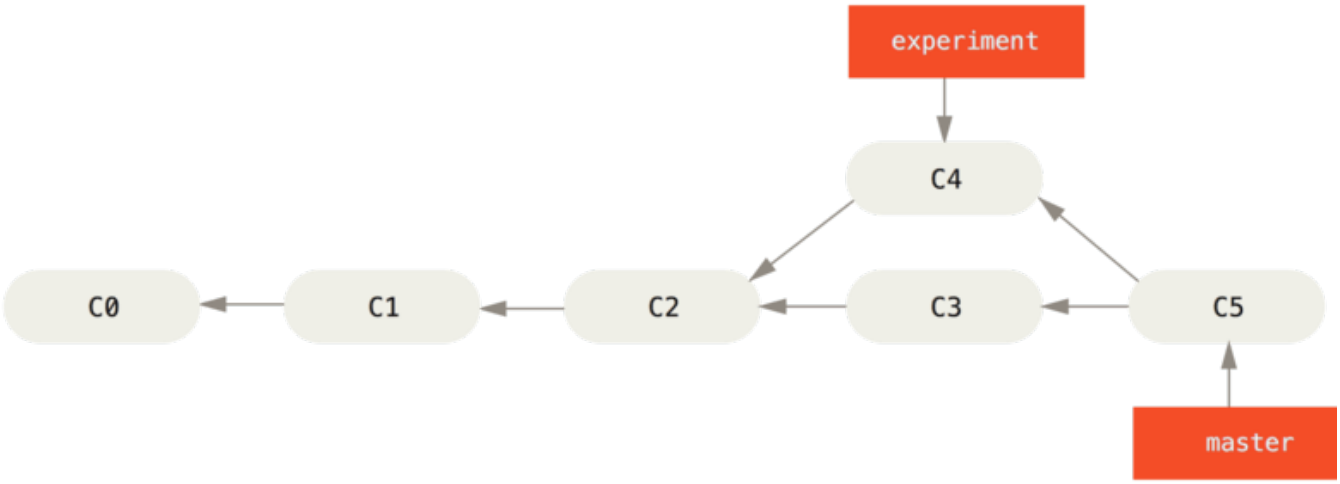



Figure 36. Merging to integrate diverged work history

However, there is another way: you can take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you'd run the following:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

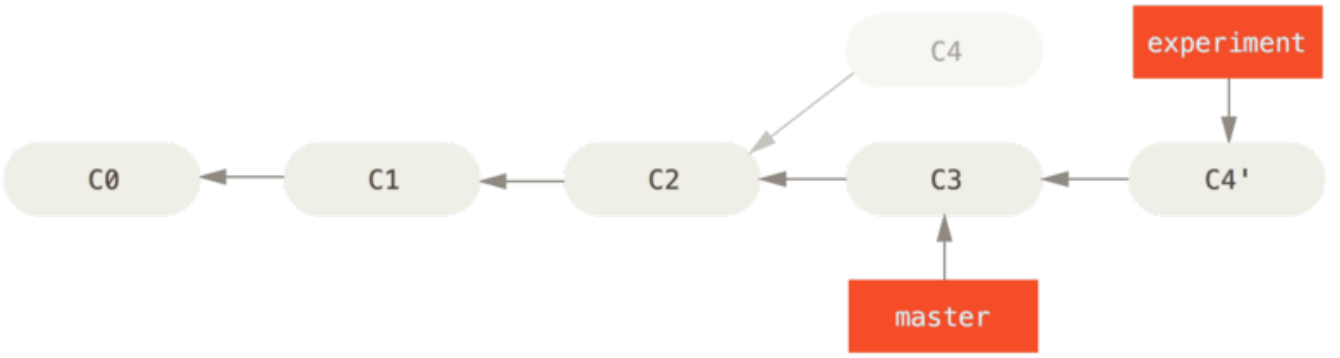



Figure 37. Rebasing the change introduced in `C4` onto `C3`

At this point, you can go back to the `master` branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```

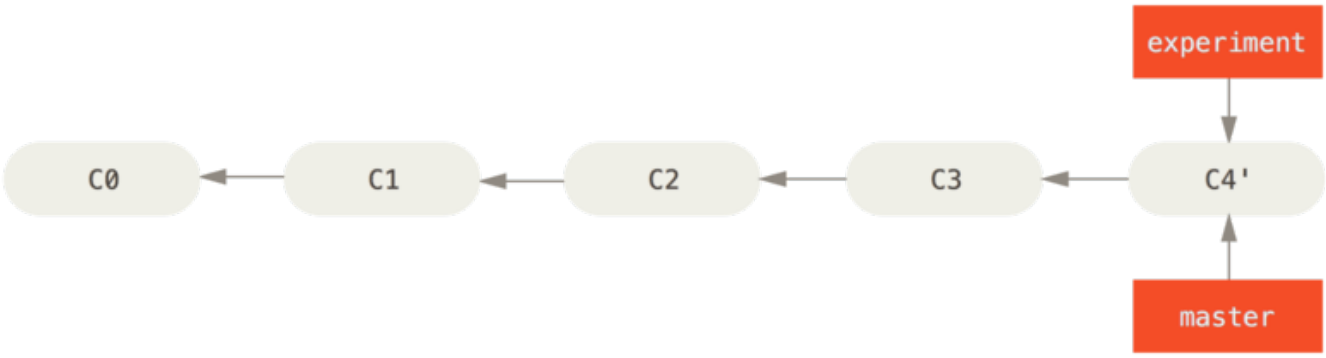



Figure 38. Fast-forwarding the master branch

Now, the snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history.

If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

### More Interesting Rebases -

https://git-scm.com/book/en/v2/Git-Branching-Rebasing#rbdiag_e

### Rebase vs. Merge

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a **record of what actually happened.** It's a historical document, valuable in its own right, and shouldn't be tampered with. From this angle, changing the commit history is almost blasphemous; you're *lying* about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made.** You wouldn't publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

**In general, the way to get the best of both worlds is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.**

**Covered Aspects till now: -**

1. Creating and switching to new branches
2. Switching between branches and merging local branches together
3. Share your branches by pushing them to a shared server
4. Working with others on shared branches and rebasing your branches before they are shared

Locally – Using Git/Source Tree
Remote Repo - Pushing to BitBucket Server(Remote Branch)

# 4.0 Branching Workflows

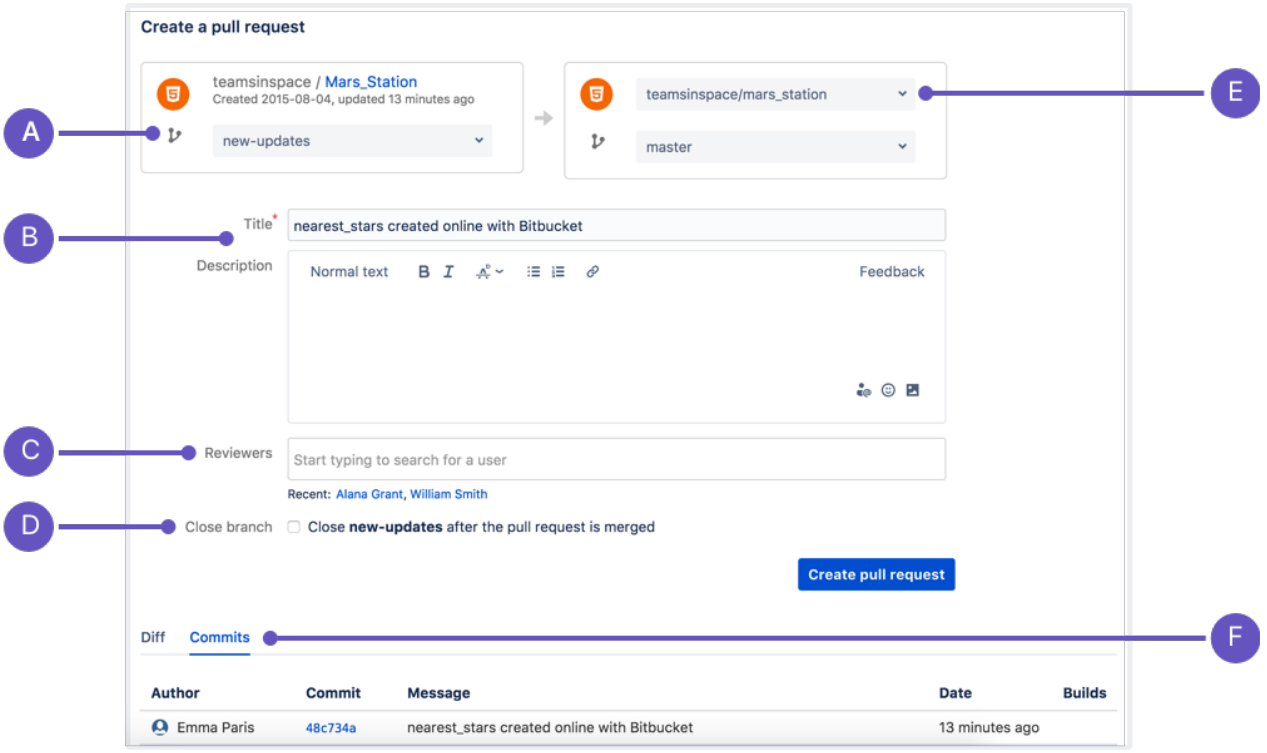
-> Master Branch – Read Only for developers. Read/Write for Admins.
-> Each developer can have multiple branches depending on issues or functionality she is working on.

1. Fetch/Pull Master branch to your system.
2. Start work by creating a branch.
3. Commit and Push the **branch** to the server(Remote Repo) as and when required.
4. Once done, generate a pull request to merge with the Master Branch of the Remote Server.
5. Start a new branch to work on different functionality/issue.

BitBucket – Remote Server based on Git.

## Create a pull request

1. Once you are done with making changes locally to your branch.
2. Pull changes, if any, from the remote master branch.
3. Merge these changes into your
4. Click **Create pull request**.
5. Select the branch with the changes you want to merge.
6. Check the destination repository and branch.
7. Add a title that can be easily recognized in notifications and the pull request list.




**A. Source:** The name of the branch you want to merge. This is the source commit, branch, or fork you are merging. You cannot modify the repository of the source but you can change the branch so check to be sure you're merging the correct branch.

**B. Title and Description**: What displays in notifications. This will be the display title in the pull request list and in notifications.

**C. Reviewers**: You can add as many as you want. Pretty much describes itself. You can also use @Mentions in comments to get someone to look at just one specific piece of code but not be responsible for reviewing the whole pull request.

**D. Close branch**: Select to automatically close the branch. When you select this the source branch will be closed automatically once the pull request is merged. The closed branch will no longer appear in your list of branches in Bitbucket.

**E. Destination**: The destination repository and branch. This is the repository, branch, or fork you are merging your change into. You can modify the path for both the repository and the branch.

**F. Diff and Commits**: A comparison of changes and relevant commits.

- Diffs: Shows a comparison of the changes in your pull request to the files in the destination branch.
- Commits: displays a list of all the relevant commits. This list will be updated with any commits to the files on the source branch until the pull request is either merged or declined.

Select reviewers for this pull request to make key contributors aware of the changes and create an effective review. Every reviewer can comment on the pull request and with a single click give their approval.

# 5.0 BEST PRACTICES

## Write Good Commit Messages

Begin your message with a short summary of your changes (up to 50 characters as a guideline).

## Commit Related Changes

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other team members to understand the changes and roll them back if something went wrong.

## Commit Often

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it's easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard both to solve conflicts and to comprehend what happened.

## Don't Commit Half-Done Work

You should only commit code when it's completed. This doesn't mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to have something in the repository before leaving the office at the end of the day.

## Test Before You Commit

Resist the temptation to commit something that you "think" is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell).

## Use Branches

Branching is one of Git's most powerful features – and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas…

## Agree on a Workflow

Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, git-flow… Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates' personal preferences. However, you choose to work, just make sure to agree on a common workflow that everyone follows.