**GIT**

## First-Time Git Setup

### [Your Identity](http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Your-Identity)

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

**$** git config --global user.name "John Doe"

**$** git config --global user.email johndoe@example.com

### [Checking Your Settings](http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Checking-Your-Settings)

**$** git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

**$** git config user.name

John Doe

## Getting a Git Repository

### [Cloning an Existing Repository](http://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository#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.

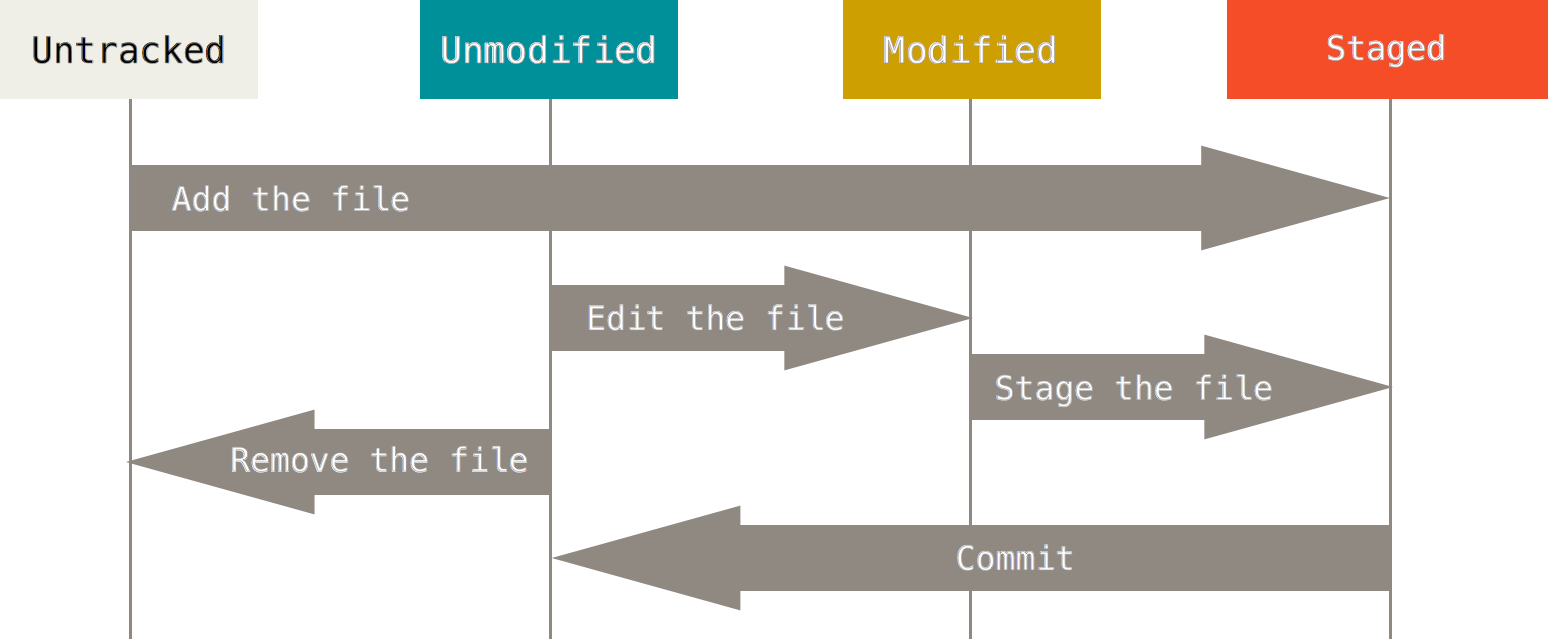
You clone a repository with git clone [url]. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

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

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

## Recording Changes to the Repository



### [Checking the Status of Your Files](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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

nothing to commit, working directory clean

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

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)

### [Tracking New Files](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

### [Staging Modified Files](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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

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

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

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

### [Ignoring Files](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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.

 In such cases, you can create a file listing patterns to match them named .gitignore.Here is an example .gitignore file:

**$** cat .gitignore

\*.[oa]

\*~

Here is another example .gitignore file:

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the root TODO file, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .txt files in the doc/ directory

doc/\*\*/\*.txt

### [Committing Your Changes](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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.

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

### [Removing Files](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#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

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.

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 that end with ~.

## Undoing Things

### [Unstaging a Staged File](http://git-scm.com/book/en/v2/Git-Basics-Undoing-Things#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

**$** 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

While git reset can be a dangerous command if you call it with --hard, in this instance the file in your working directory is not touched. Calling git reset without an option is not dangerous - it only touches your staging area.

### [Unmodifying a Modified File](http://git-scm.com/book/en/v2/Git-Basics-Undoing-Things#Unmodifying-a-Modified-File)

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

**$** git checkout -- CONTRIBUTING.md

**$** git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

## Working with Remotes

### [Showing Your Remotes](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Showing-Your-Remotes)

**$** git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

**$** cd ticgit

**$** git remote

origin

**$** git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

### [Adding Remote Repositories](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#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)

### [Fetching and Pulling from Your Remotes](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Fetching-and-Pulling-from-Your-Remotes)

to get data from your remote projects, you can run:

**$** git fetch [remote-name]

**$** 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

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command pulls 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.

If you have a branch set up to track a remote, you can use the git pull command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

### [Pushing to Your Remotes](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#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]. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

**$** git push origin master

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to pull down their work first and incorporate it into yours before you’ll be allowed to push.

### [Inspecting a Remote](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#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':

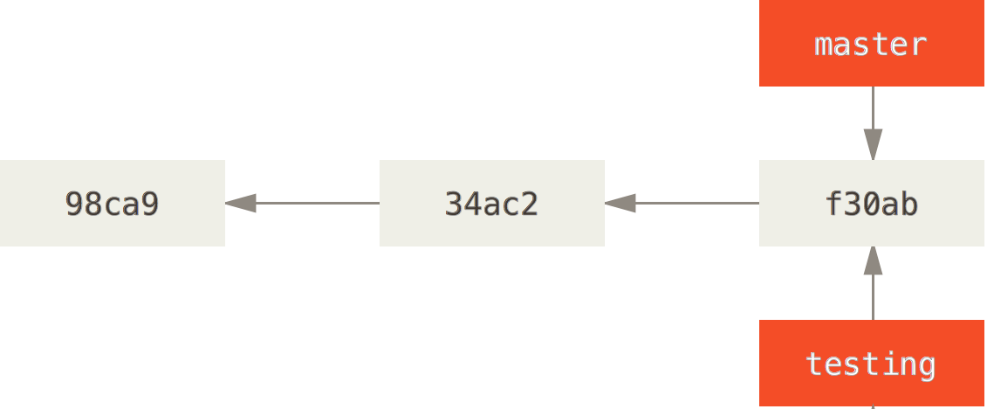
master pushes to master (up to date)

## Branches in a Nutshell

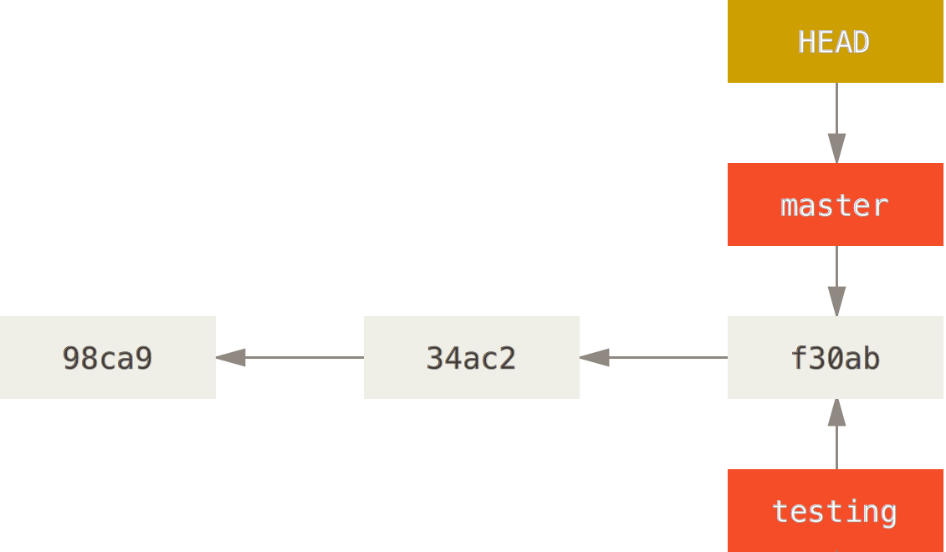
### [Creating a New Branch](http://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#Creating-a-New-Branch)

**$** git branch testing

This creates a new pointer at the same commit you’re currently on.



How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only created a new branch – it didn’t switch to that branch.



### [Switching Branches](http://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#Switching-Branches)

**$** git checkout testing

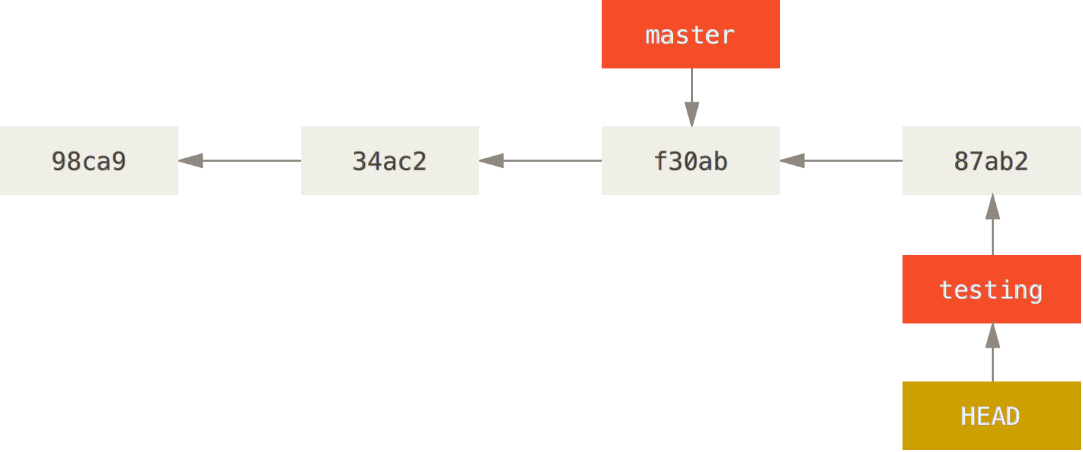
This moves HEAD to point to the testing branch.

### HEAD points to the current branch.

Well, let’s do another commit:

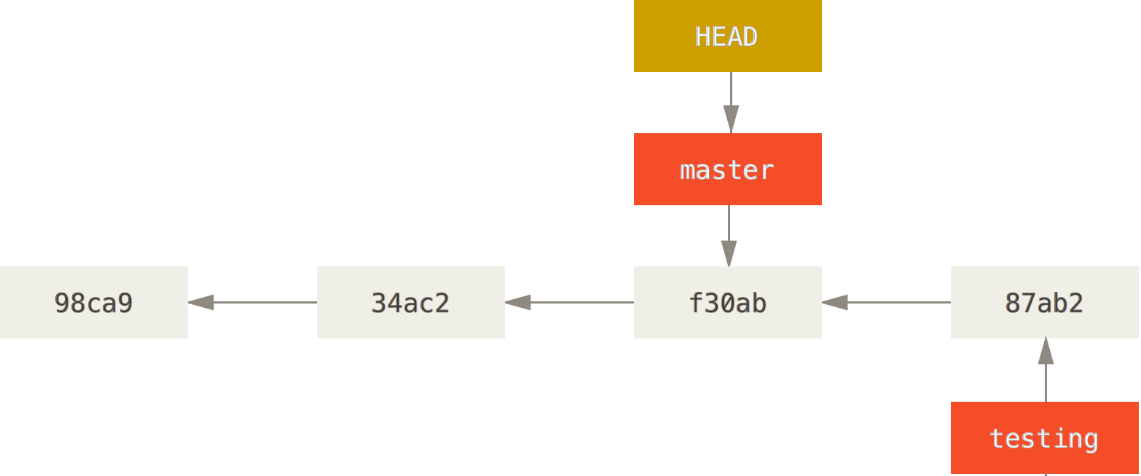
**$** vim test.rb

**$** 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



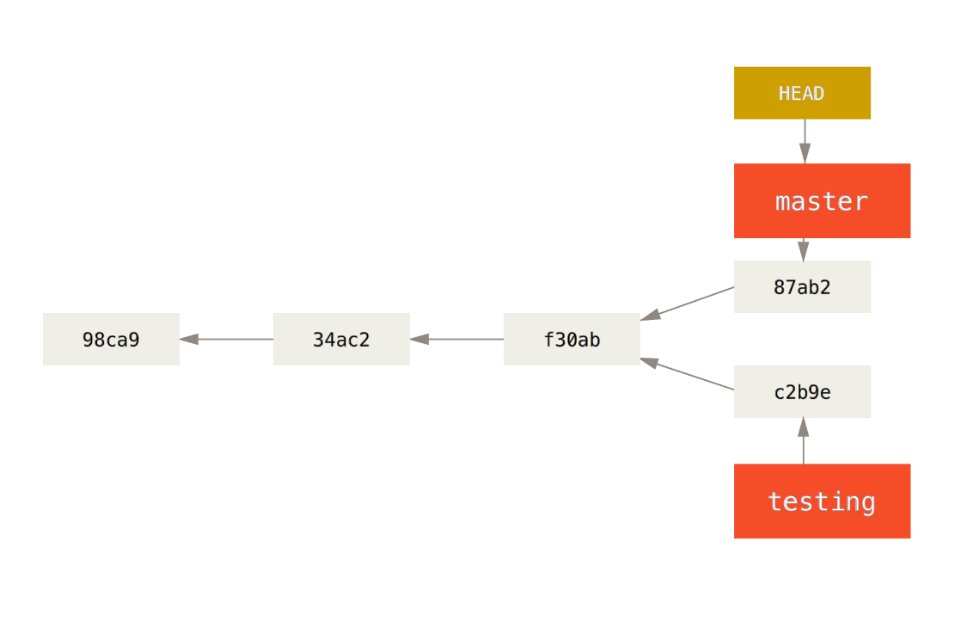
That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.

Let’s make a few changes and commit again:

**$** vim test.rb

**$** git commit -a -m 'made other changes'

You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. 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, andcommit commands.



## 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:

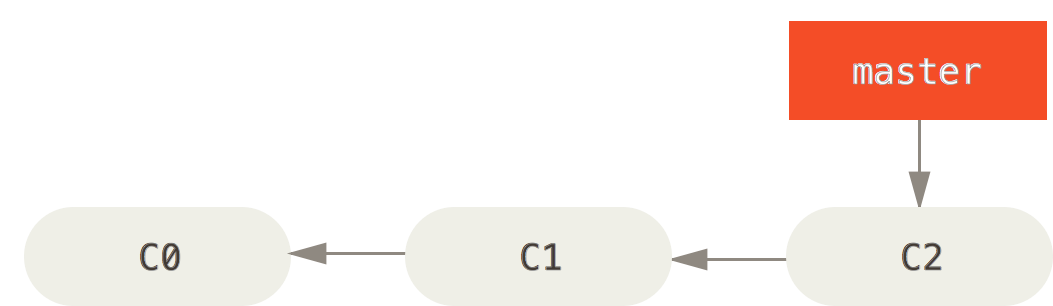
1. Do work on a web site.
2. Create a branch for a new story you’re working on.
3. Do some work in that branch.

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](http://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#Basic-Branching)

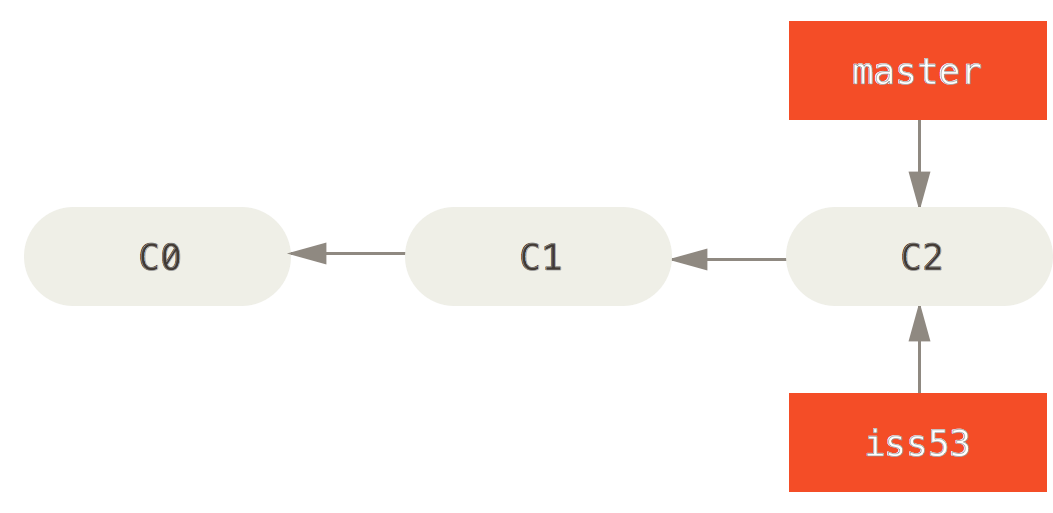
First, let’s say you’re working on your project and have a couple of commits already.



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

**$** git checkout -b iss53

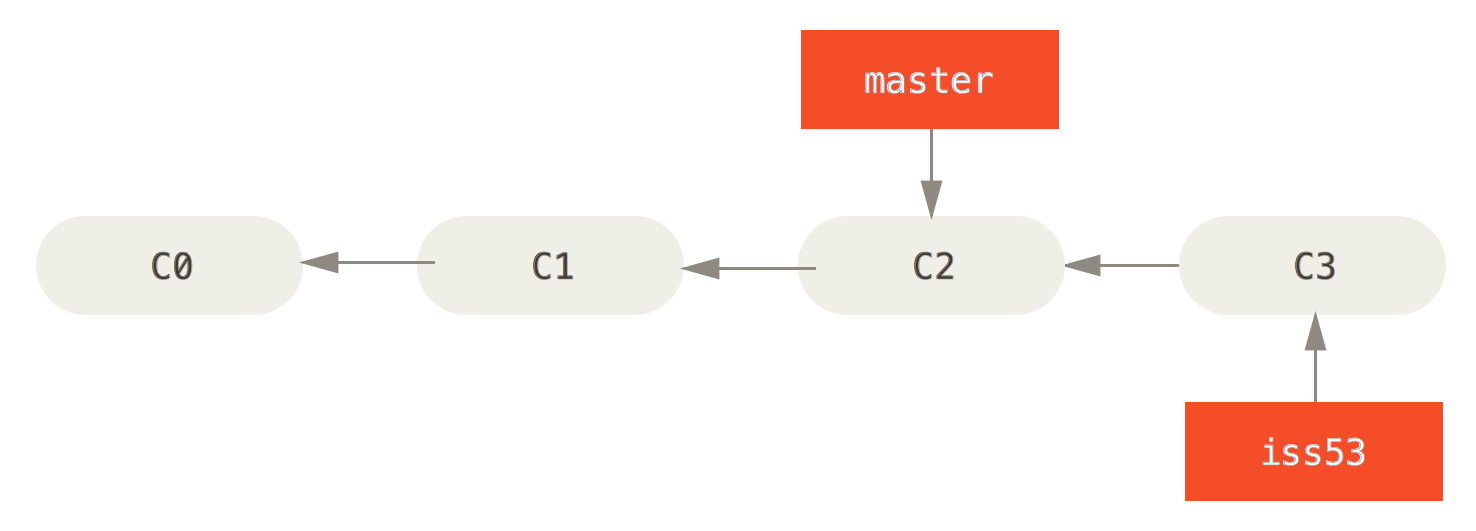
Switched to a new branch "iss53"



You work on your web site 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]'



Now you get the call that there is an issue with the web site, and you need to fix it immediately.All you have to do is switch back to your master branch.

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'

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.

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

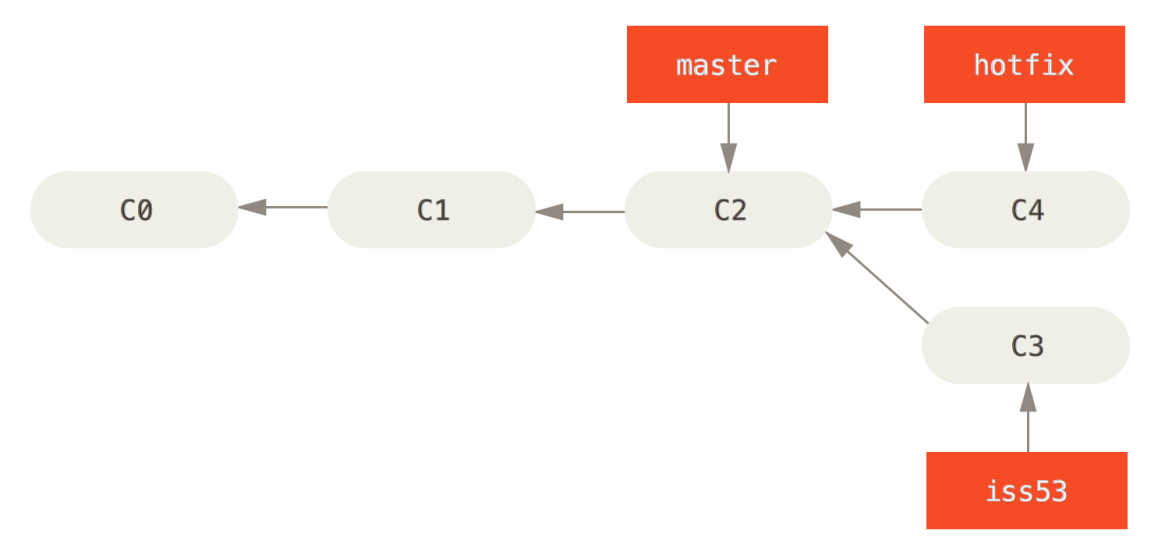
Switched to a new branch 'hotfix'

**$** vim index.html

**$** git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)



Make sure the hotfix is what you want, and merge it back into your master branch to deploy to production. You do this with the git merge command:

**$** git checkout master

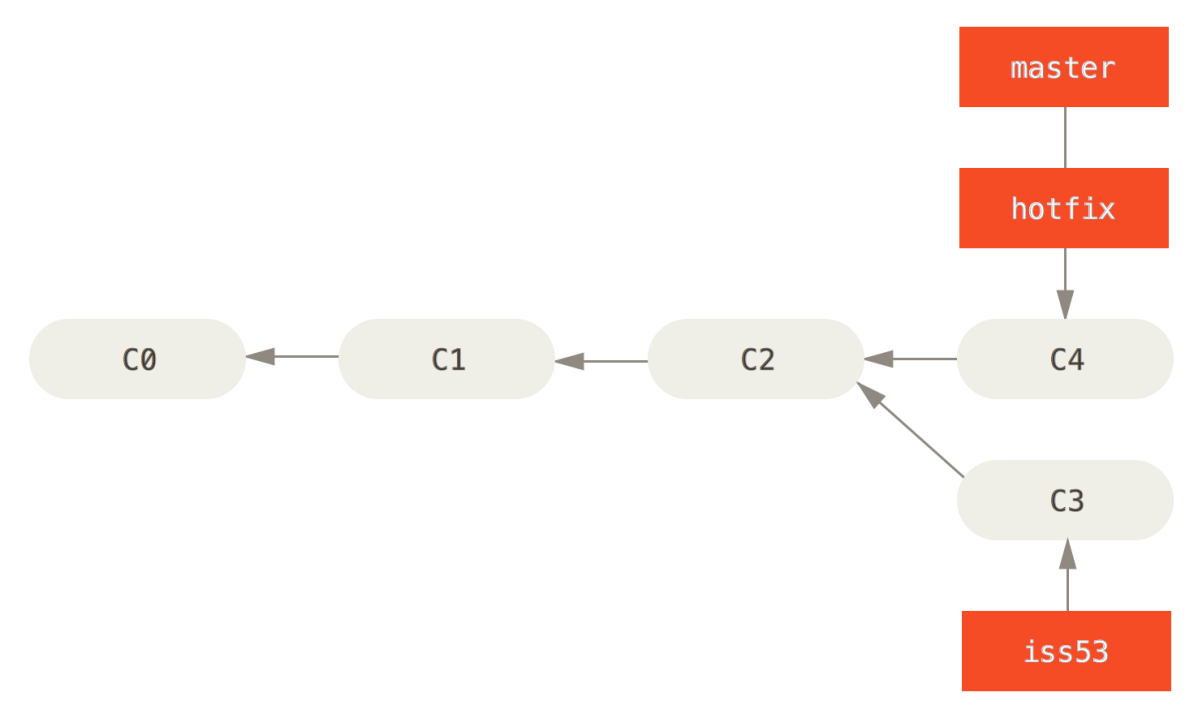
**$** git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)



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

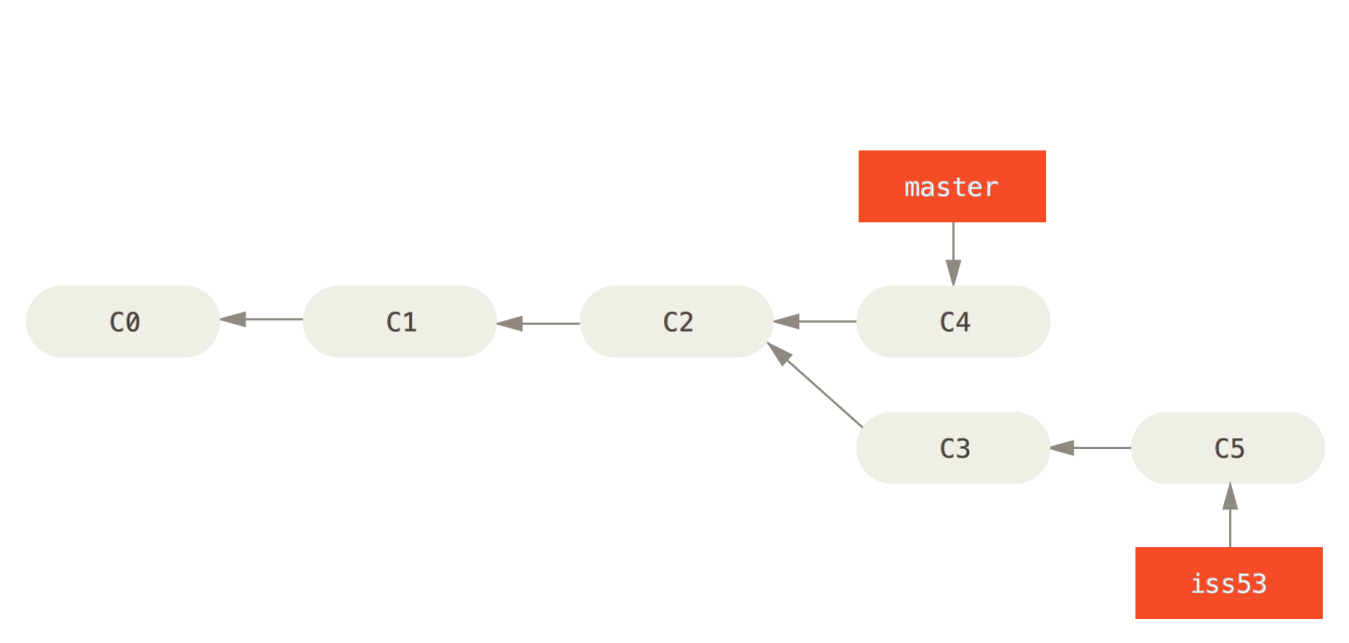
Switched to branch "iss53"

**$** 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](http://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#Basic-Merging)

Suppose you’ve decided that your issue #53 work is complete and ready to be merged into yourmaster branch. In order to do that, you’ll merge in your iss53 branch, much like you merged in yourhotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

**$** git checkout master

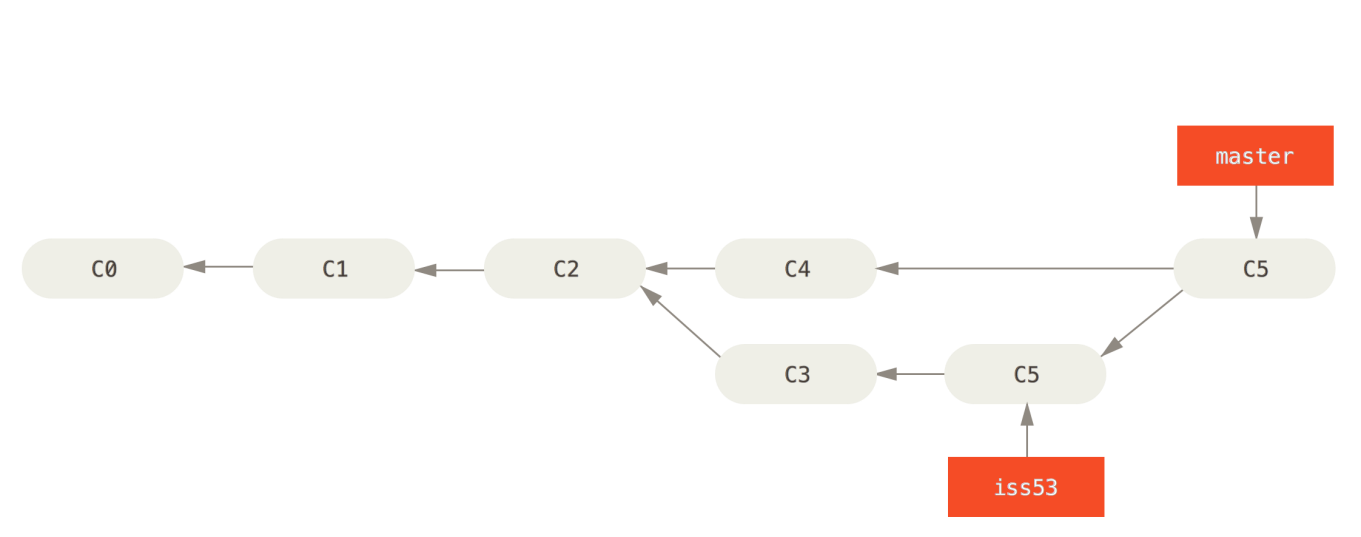
Switched to branch 'master'

**$** git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)



### [Basic Merge Conflicts](http://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#Basic-Merge-Conflicts)

**$** 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 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

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.

## Branch Management

Listing of your current branches:

**$** git branch

iss53

\* master

testing

Notice the \* character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to). This means that if you commit at this point, themaster branch will be moved forward with your new work. To see the last commit on each branch, you can run git branch -v:

**$** git branch -v

iss53 93b412c fix javascript issue

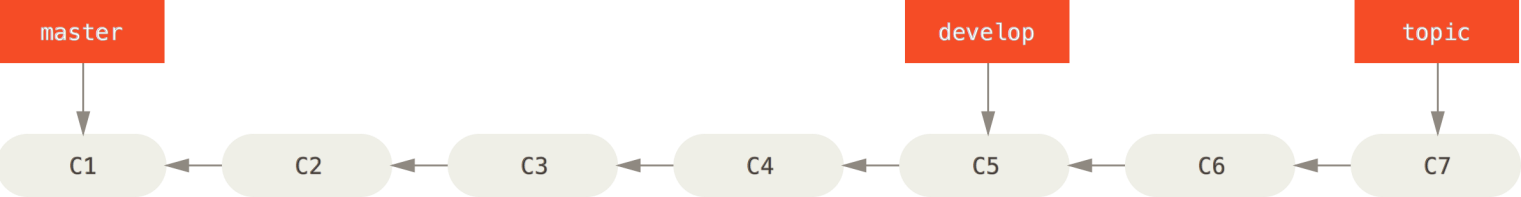
\* master 7a98805 Merge branch 'iss53'

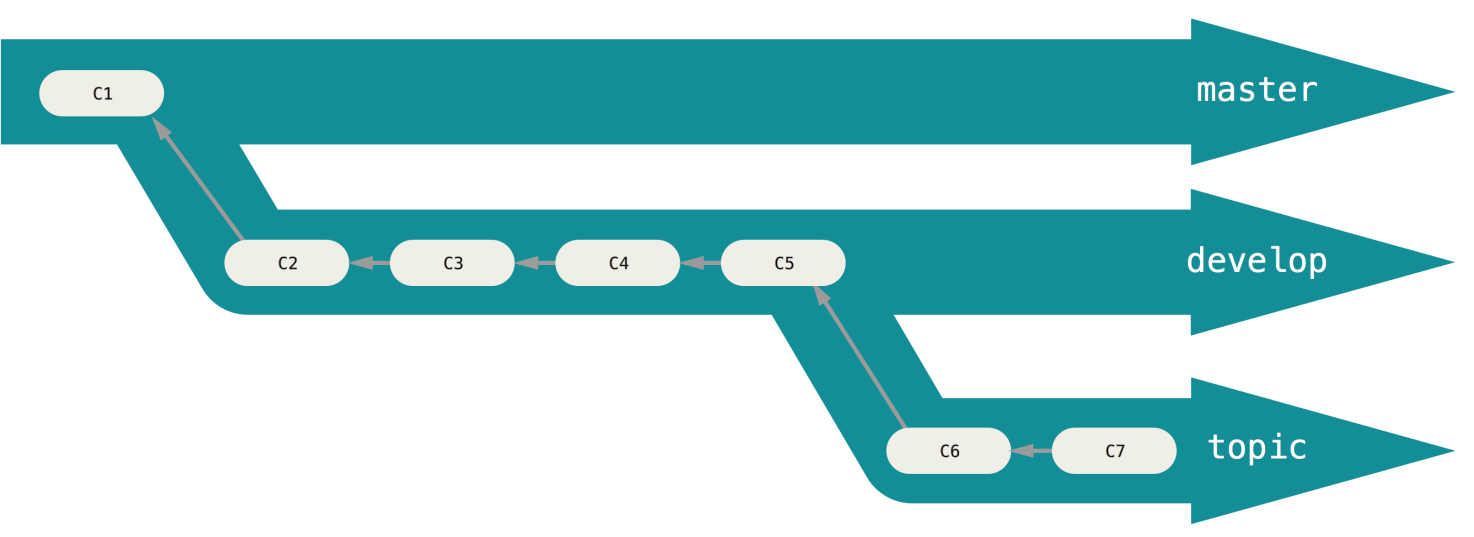
testing 782fd34 add scott to the author list in the readmes

## Branching Workflows

### [Long-Running Branches](http://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows#Long-Running-Branches)

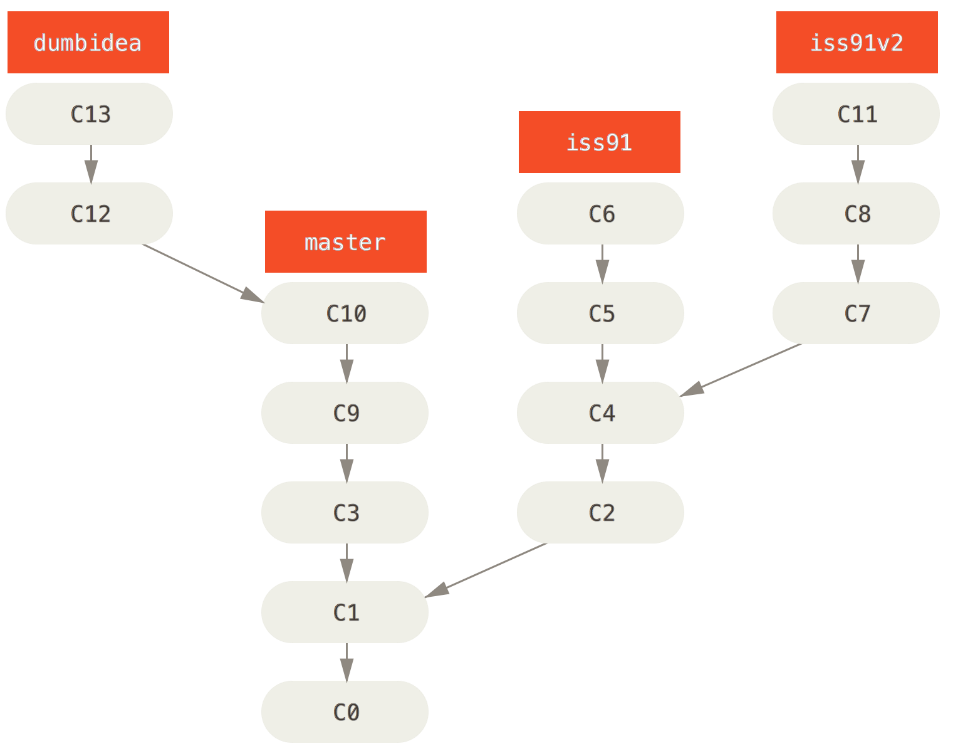
The idea is that your branches are at various levels of stability; when they reach a more stable level, they’re merged into the branch above them. Again, having multiple long-running branches isn’t necessary, but it’s often helpful, especially when you’re dealing with very large or complex projects.

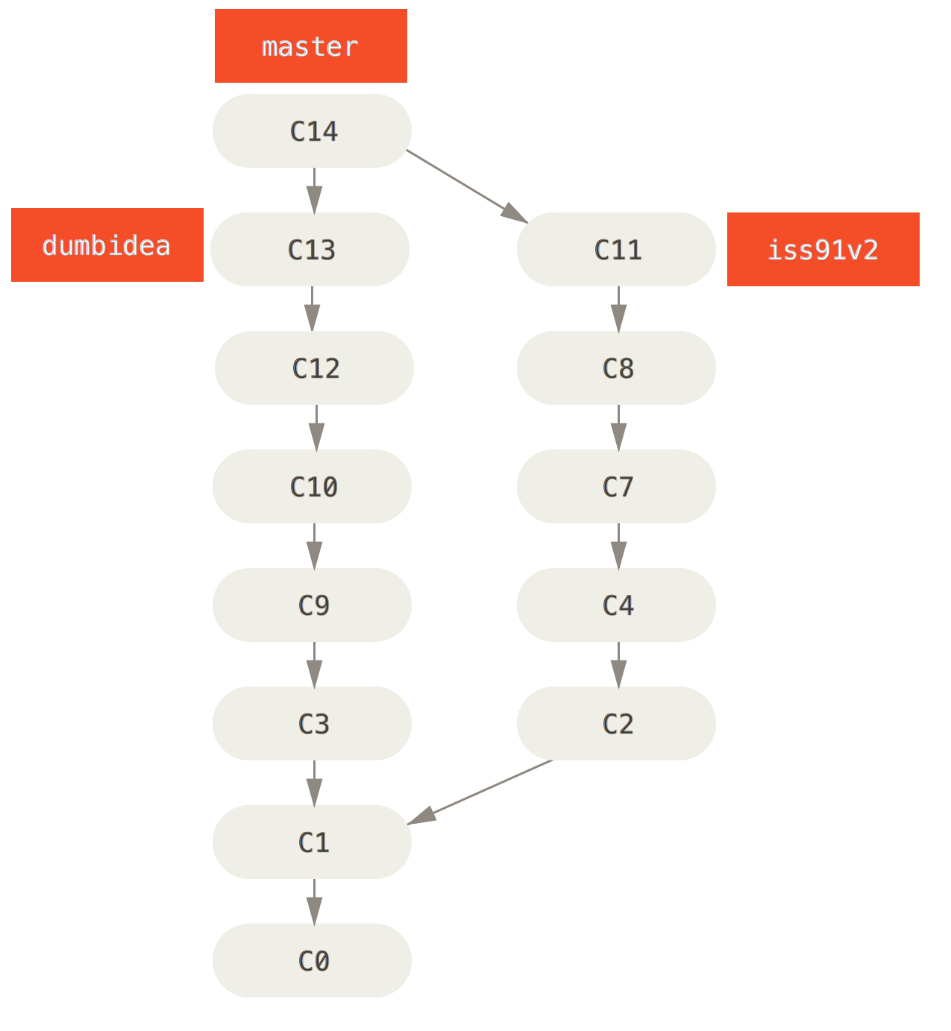




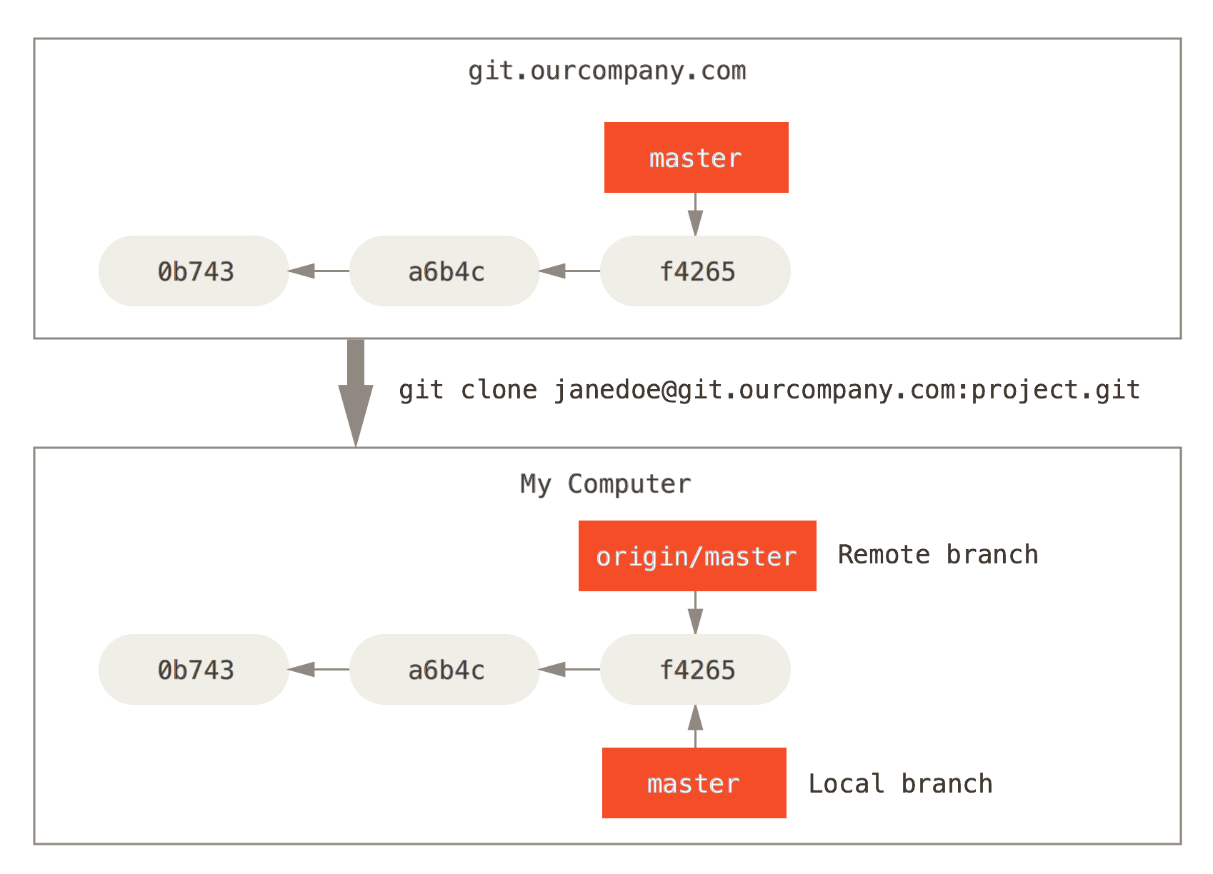
### [Topic Branches](http://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows#Topic-Branches)

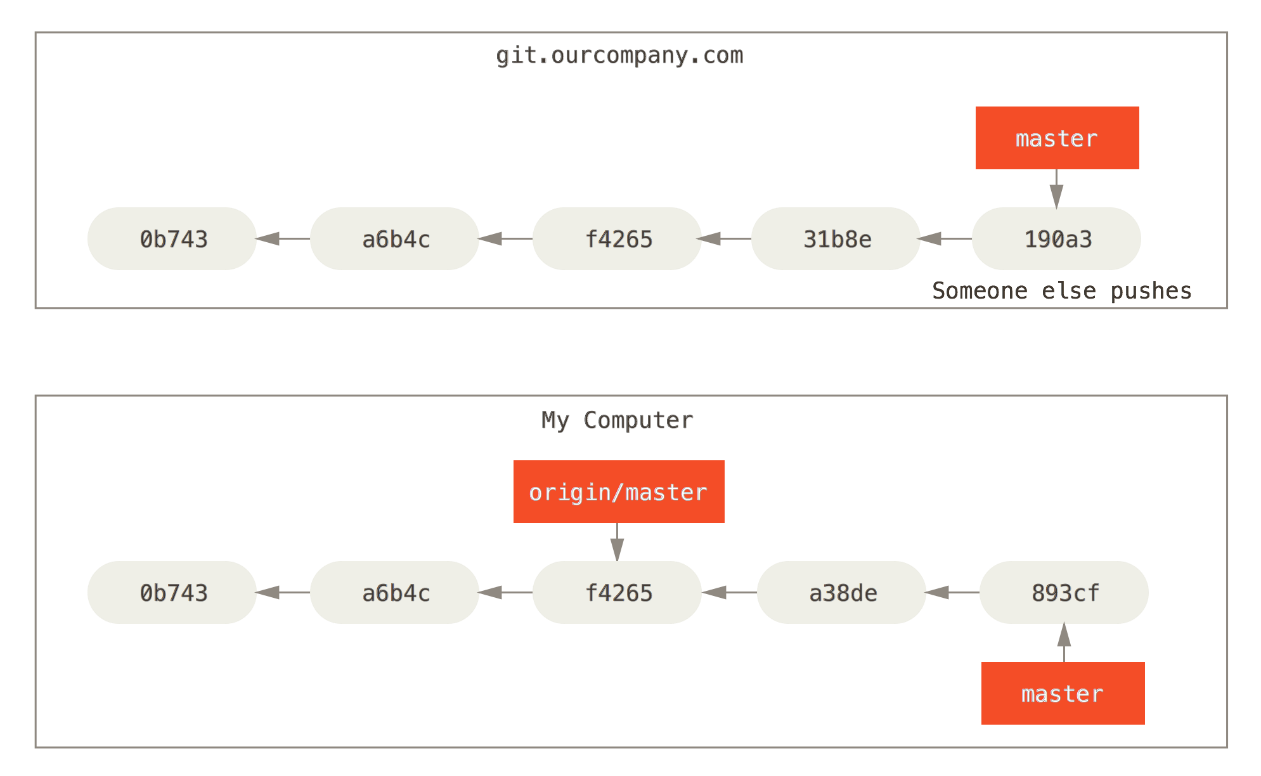
Topic branches are useful in projects of any size.



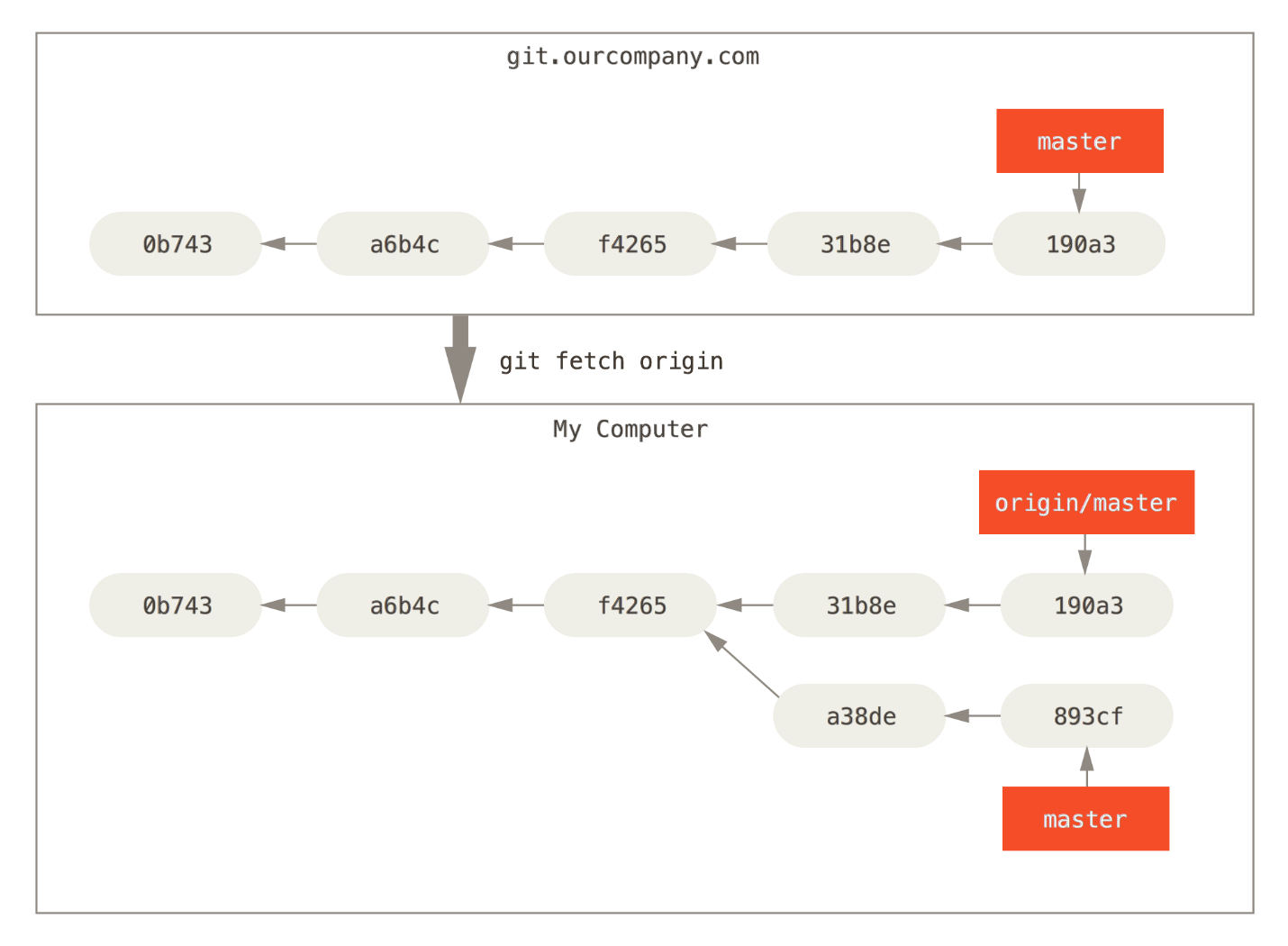


## Remote Branches





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.



### [Pushing](http://git-scm.com/book/en/v2/Git-Branching-Remote-Branches#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.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. 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

You can also do git push origin serverfix:serverfix, which does the same thing.

If you didn’t want it to be called serverfix on the remote, you could instead run git push origin serverfix:awesomebranch to push your local serverfix branch to theawesomebranch branch on the remote project.

### [Tracking Branches](http://git-scm.com/book/en/v2/Git-Branching-Remote-Branches#Tracking-Branches)

Checking out a local branch from a remote branch automatically creates what is called a “tracking branch” (or sometimes 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 tracksorigin/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'

**$** git checkout -b serverfix origin/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'

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 togit branch to explicitly set it at any time.

**$** git branch -u origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

## Upstream shorthand

When you have a tracking branch set up, you can reference it with the @{upstream} or @{u}shorthand. So if you’re on the master branch and it’s tracking origin/master, you can say something like git merge @{u} instead of git merge origin/master if you wish.

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

### [Pulling](http://git-scm.com/book/en/v2/Git-Branching-Remote-Branches#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 fetchimmediately followed by a git merge in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by theclone or checkout commands, git pull will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it’s better to simply use the fetch and merge commands explicitly as the magic of git pull can often be confusing.

### [Deleting Remote Branches](http://git-scm.com/book/en/v2/Git-Branching-Remote-Branches#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.

You can delete a remote branch using the --delete option to git push. If you want to delete yourserverfix branch from the server, you run the following:

**$** git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix