**GIT**

1. Cấu hình git lần đầu

Việc đầu tiên bạn nên làm khi cấu hình Git là chỉ định tên tài khoản và địa chỉ e-mail. Điều này rất quan trọng vì mỗi Git sẽ sử dụng chúng cho mỗi lần commit, những thông tin này được gắn bất di bất dịch vào các commit:

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

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

Liệt kê tất cả các cài đặt của Git:

$ git config --list

1. Sao chép kho chứa có sẵn

Để sao chép một kho chứa:

$ git clone git://github.com/schacon/grit.git

 Sao chép kho chứa này vào một thư mục có tên khác:

$ git clone git://github.com/schacon/grit.git mygrit

1. Thay đổi kho chứa

Ban đầu, khi bạn tạo bản sao của một kho chứa, tất cả tập tin ở trạng thái "đã được theo dõi" (tracked) và "chưa thay đổi" (unmodified) vì bạn vừa mới tải chúng về và chưa thực hiện bất kỳ thay đổi nào.

Khi bạn chỉnh sửa các tập tin, Git coi là chúng đã bị thay đổi so với lần commit trước đó. Bạn stage các tập tin bị thay đổi này và sau đó commit tất cả các thay đổi đã được staged (tổ chức) đó, và quá trình này cứ thế lặp đi lặp lại.



**Để xem trạng thái của tập tin :**

$ git status

# On branch master

nothing to commit, working directory clean

sau khi bạn thêm mới 1 tập tin 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)

**Stage tập tin thành staged :**

$ git add README

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

đôi khi sửa 1 tập tin

$ 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)

#

# modified: benchmarks.rb

Stage chúng

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

# modified: benchmarks.rb

**Xóa tập tin :**

$ rm grit.gemspec

$ git status

# On branch master

#

# Changes not staged for commit:

# (use "git add/rm <file>..." to update what will be committed)

#

# deleted: grit.gemspec

Stage tập tin muốn xóa:

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

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

#

# deleted: grit.gemspec

Xóa tập tin trong thư mục log có đuôi .log:

$ git rm log/\\*.log

xoá toàn bộ tập tin kết thúc bằng ~

$ git rm \\*~

1. Xem lịch sử commit

Xem lịch sử commit bằng giao diện:

$ gitk

1. Phục hồi

**Unstage các tập tin staged:**

$ git add .

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

# modified: benchmarks.rb

$ git reset HEAD benchmarks.rb

benchmarks.rb: locally modified

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

#

# 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: benchmarks.rb

#

**Phục hồi tập tin đã thay đổi:**

# 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: benchmarks.rb

#

$ git checkout -- benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

1. Làm việc từ xa

Đến một thời điểm nào đó bạn muốn chia sẻ dự án của bạn, bạn phải đẩy ngược nó lên. Câu lệnh để thực hiện rất đơn giản:

git push [tên-máy-chủ] [tên-nhánh]

 Nếu bạn muốn đẩy nhánh master vào nhánh orgin trên máy chủ (nhắc lại, khi sao chép Git thường cài đặt/cấu hình mặc định các tên đó cho bạn), bạn có thể chạy lệnh sau để đẩy các công việc đã hoàn thành ngược lại máy chủ:

$ git push origin master

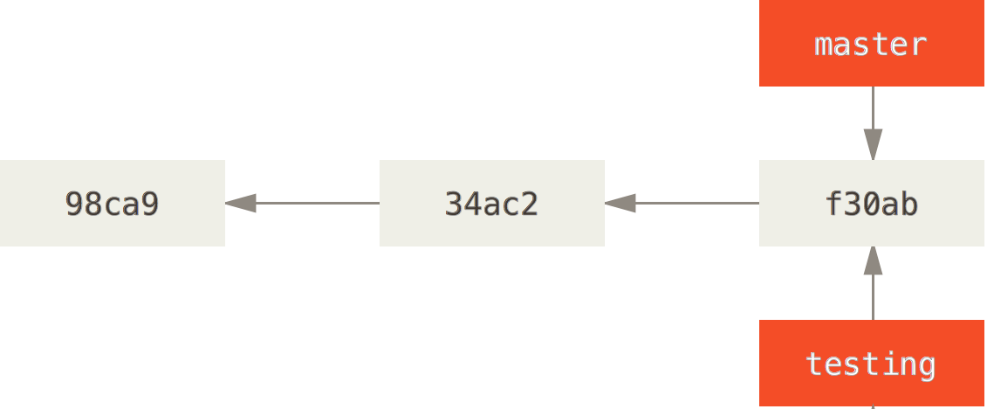
Nếu bạn và ai khác cùng sao chép tại cùng một thời điểm; người kia đẩy ngược lên, sau đó bạn cũng muốn đẩy lên, thì hành động của bạn sẽ bị từ chối ngay tức khắc. Trước hết bạn phải thực hiện kéo các thay đổi mà người đó đã thực hiện và tích hợp/gộp nó vào của bạn, sau đó bạn mới được phép đẩy lên.

## Branches in a Nutshell

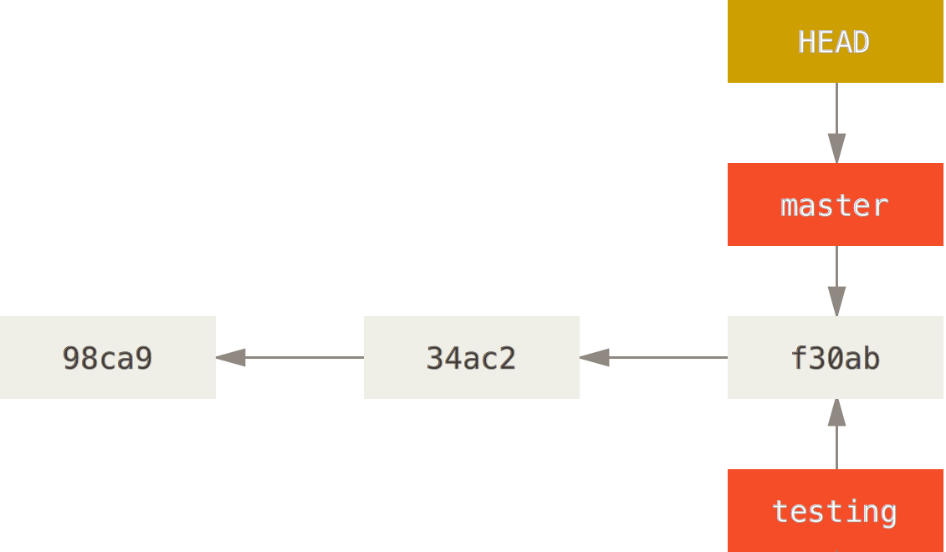
### [Creating a New Branch](http://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#Creating-a-New-Branch)

**$** 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

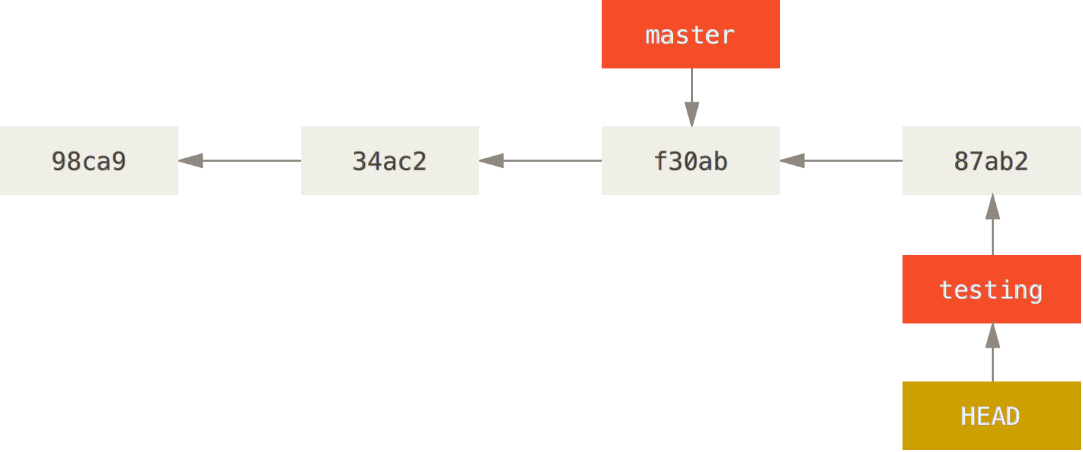
This moves HEAD to point to the testing branch.

### HEAD points to the current branch.

Well, let’s do another commit:

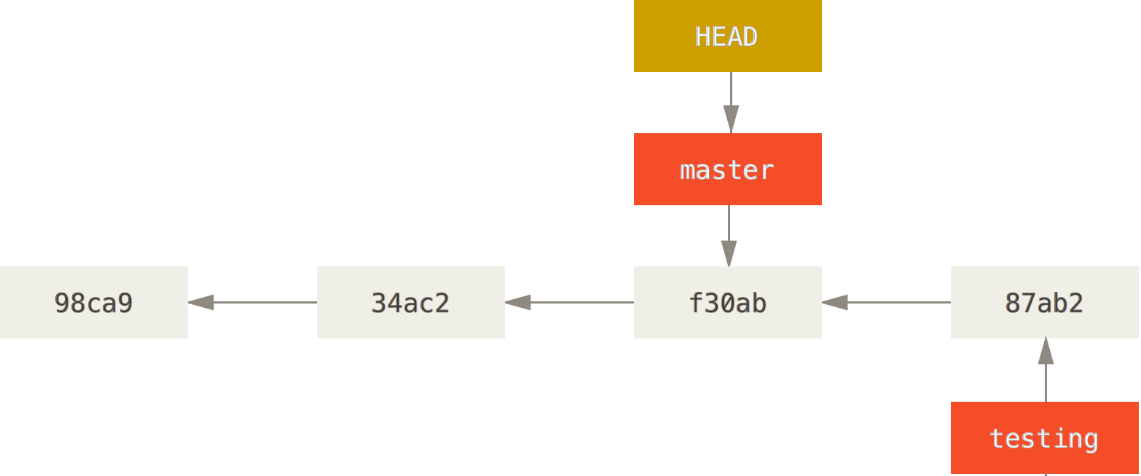
**$** vim test.rb

**$** 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



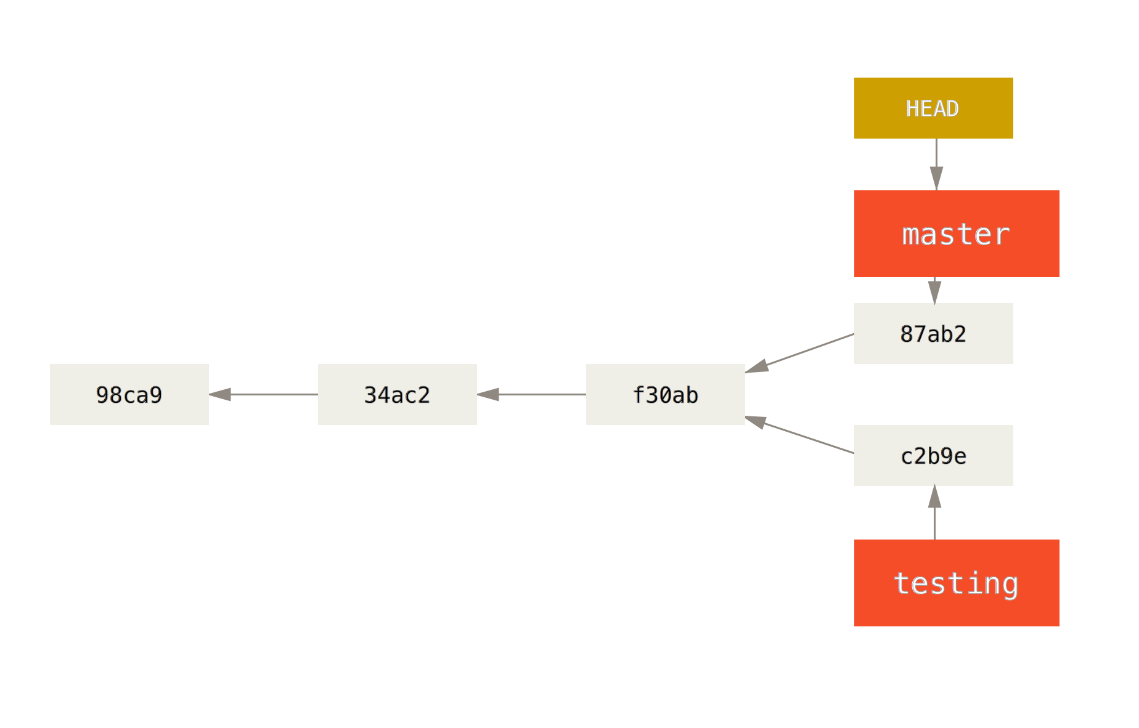
That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.

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:

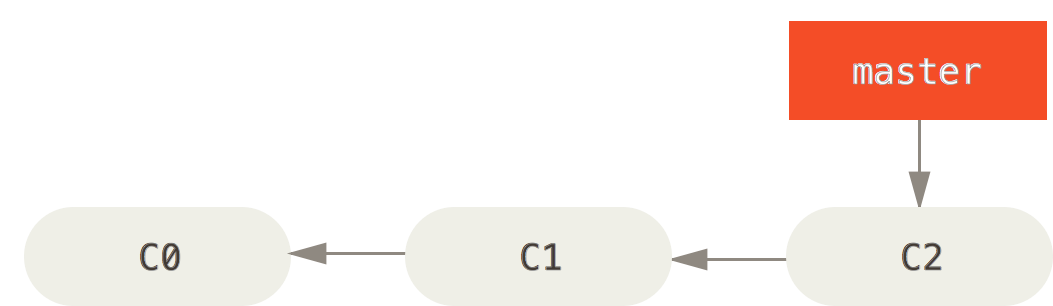
1. Do work on a web site.
2. Create a branch for a new story you’re working on.
3. Do some work in that branch.

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)

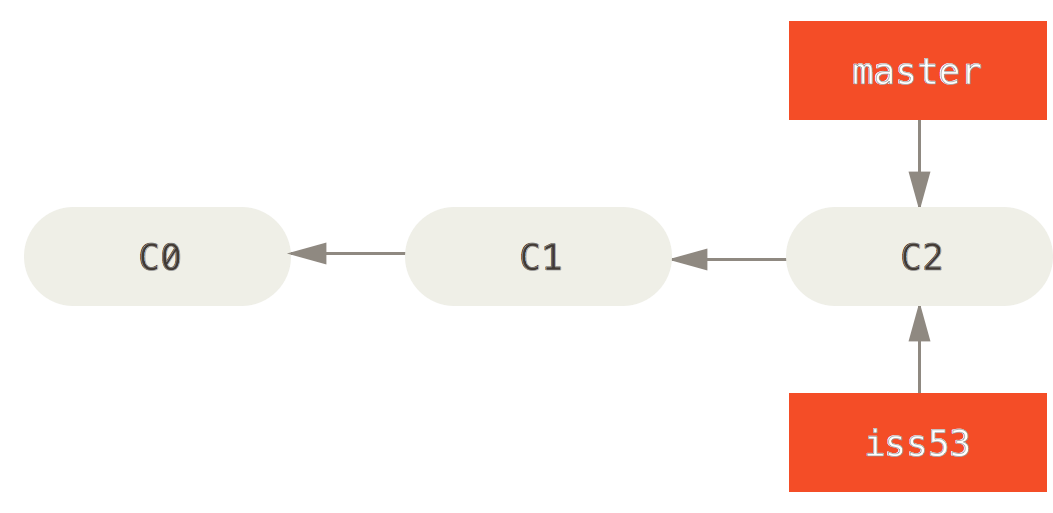
First, let’s say you’re working on your project and have a couple of commits already.



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

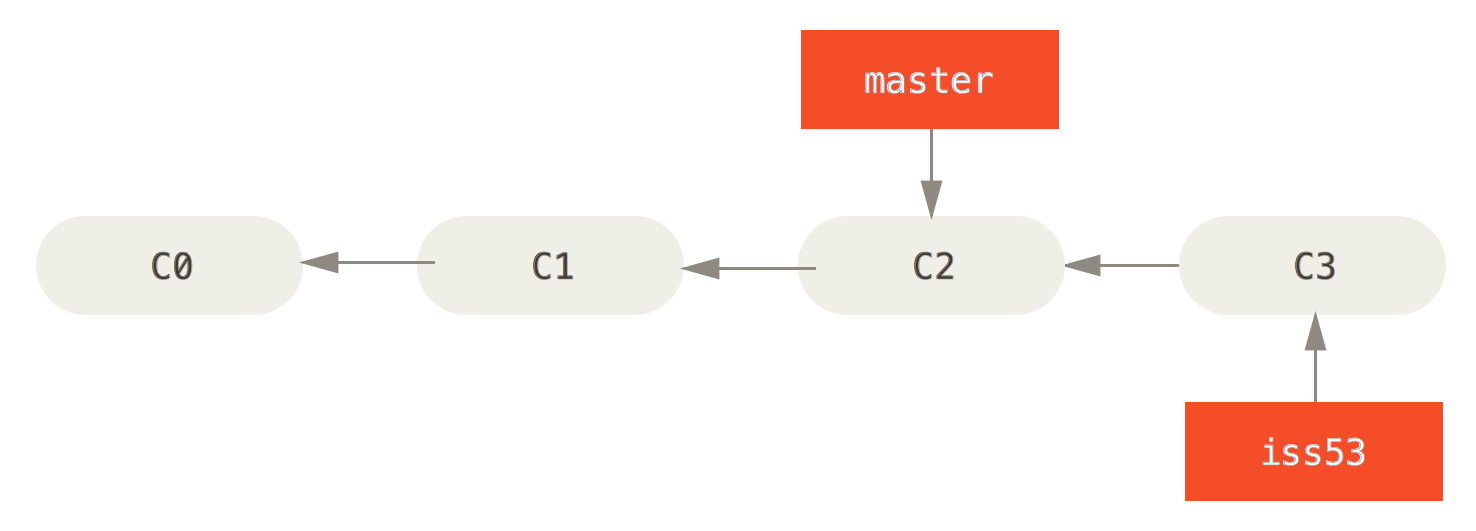
Switched to a new branch "iss53"



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

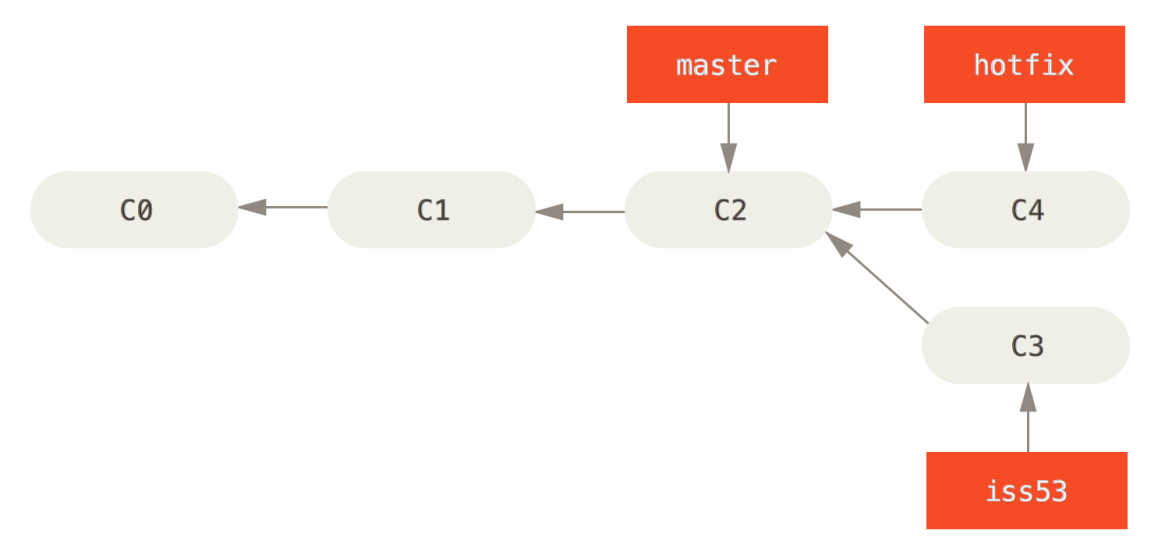
Switched to a new branch 'hotfix'

**$** 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

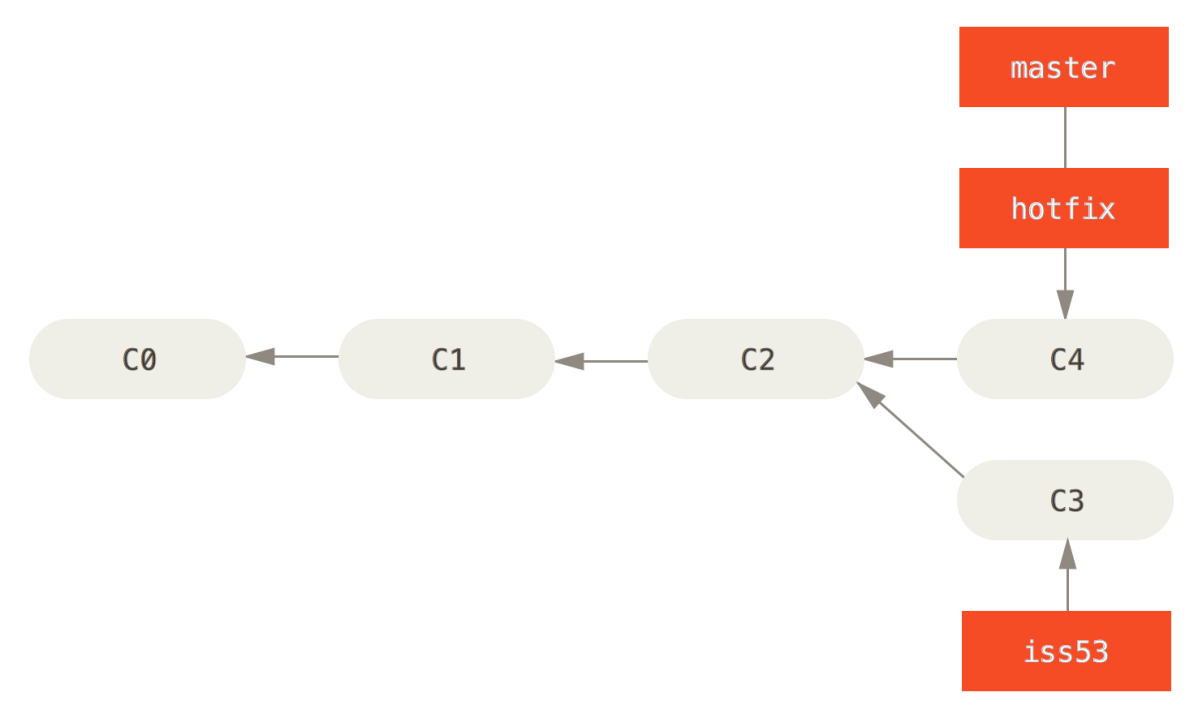
**$** git merge hotfix

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

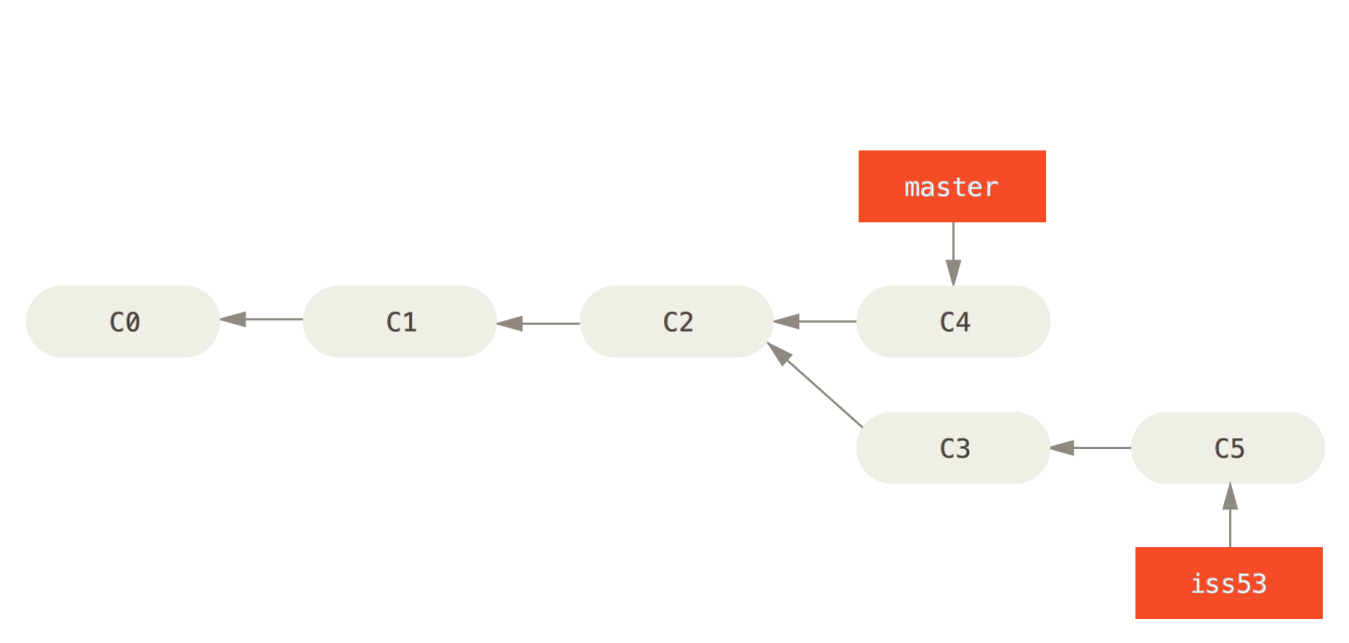
Switched to branch "iss53"

**$** 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

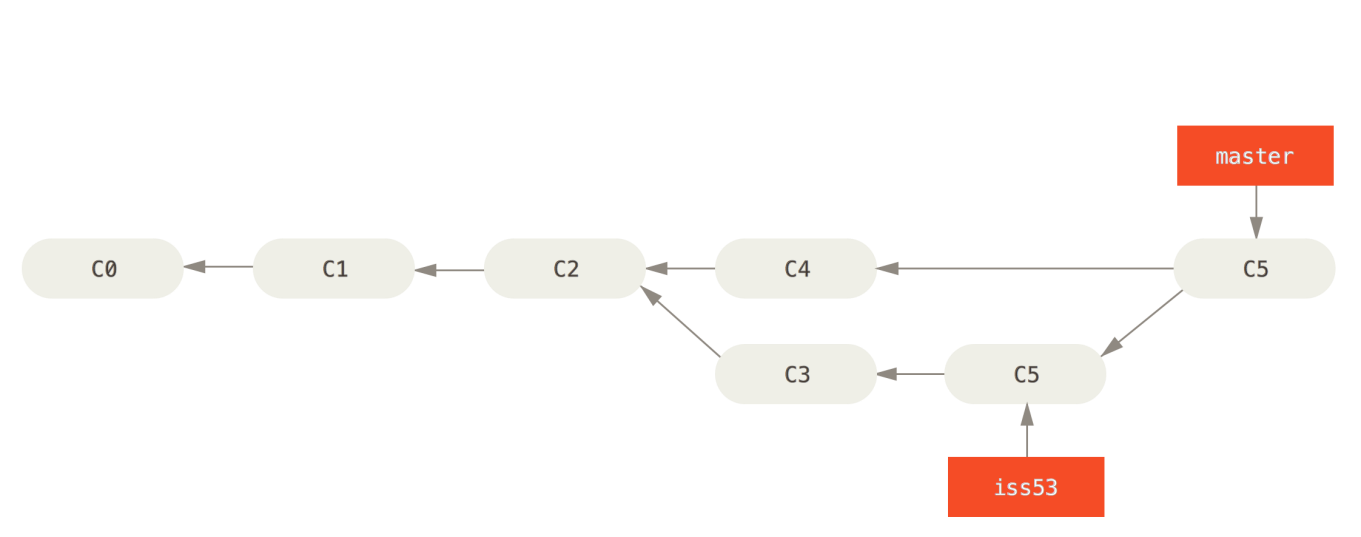
Switched to branch 'master'

**$** 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**

<divid="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

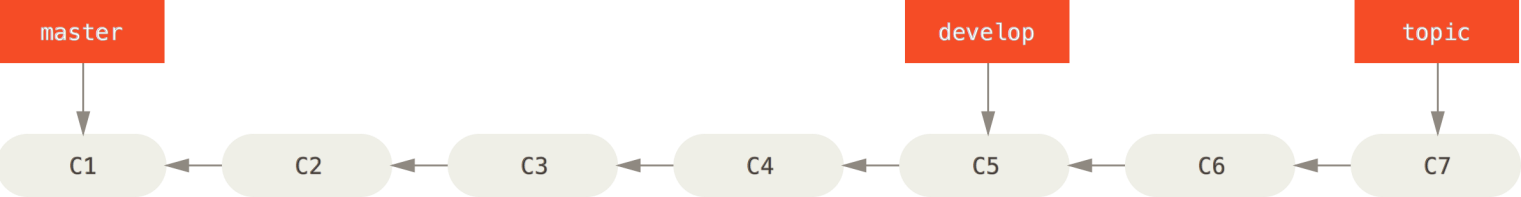
\* master 7a98805 Merge branch 'iss53'

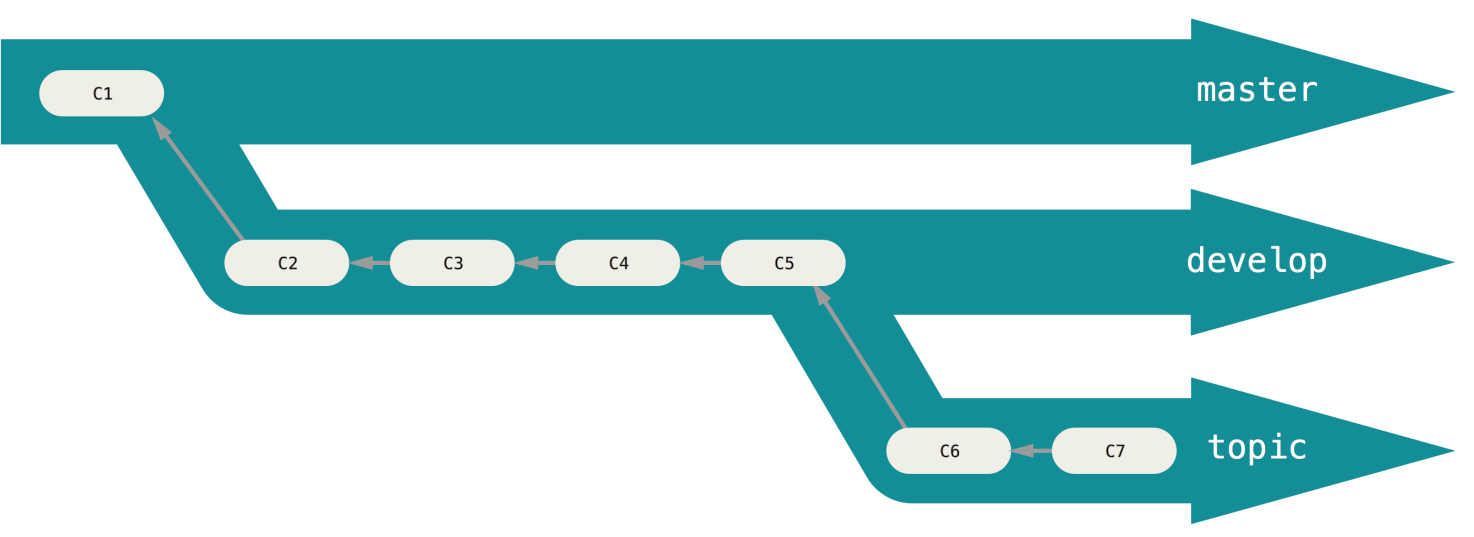
testing 782fd34 add scott to the author list in the readmes

## Branching Workflows

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

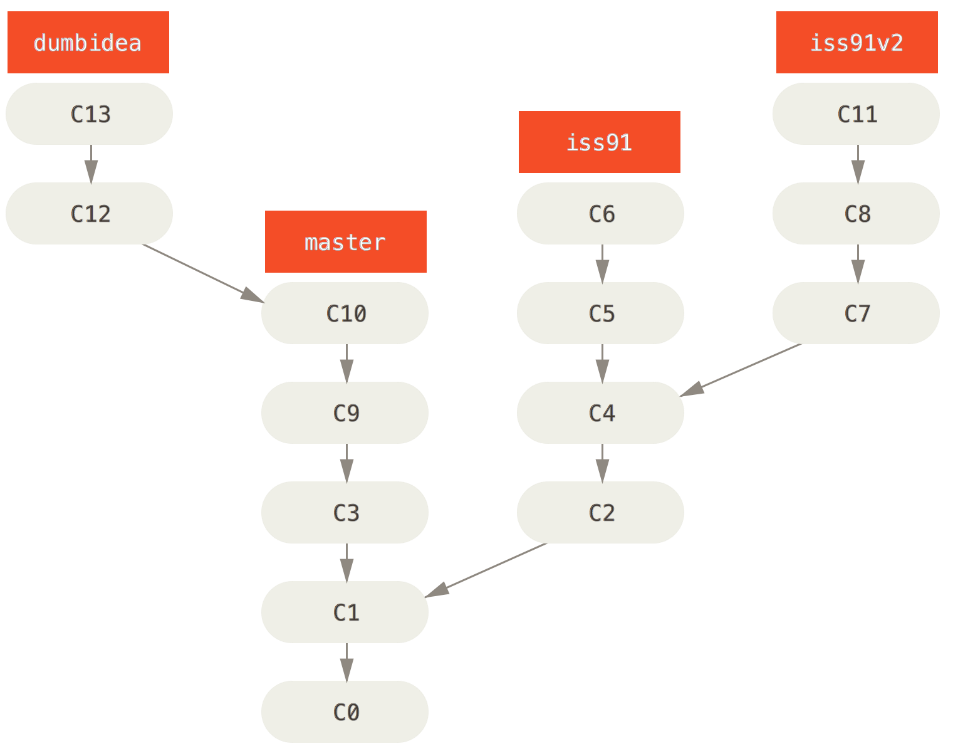
The idea is that your branches are at various levels of stability; when they reach a more stable level, they’re merged into the branch above them. Again, having multiple long-running branches isn’t necessary, but it’s often helpful, especially when you’re dealing with very large or complex projects.

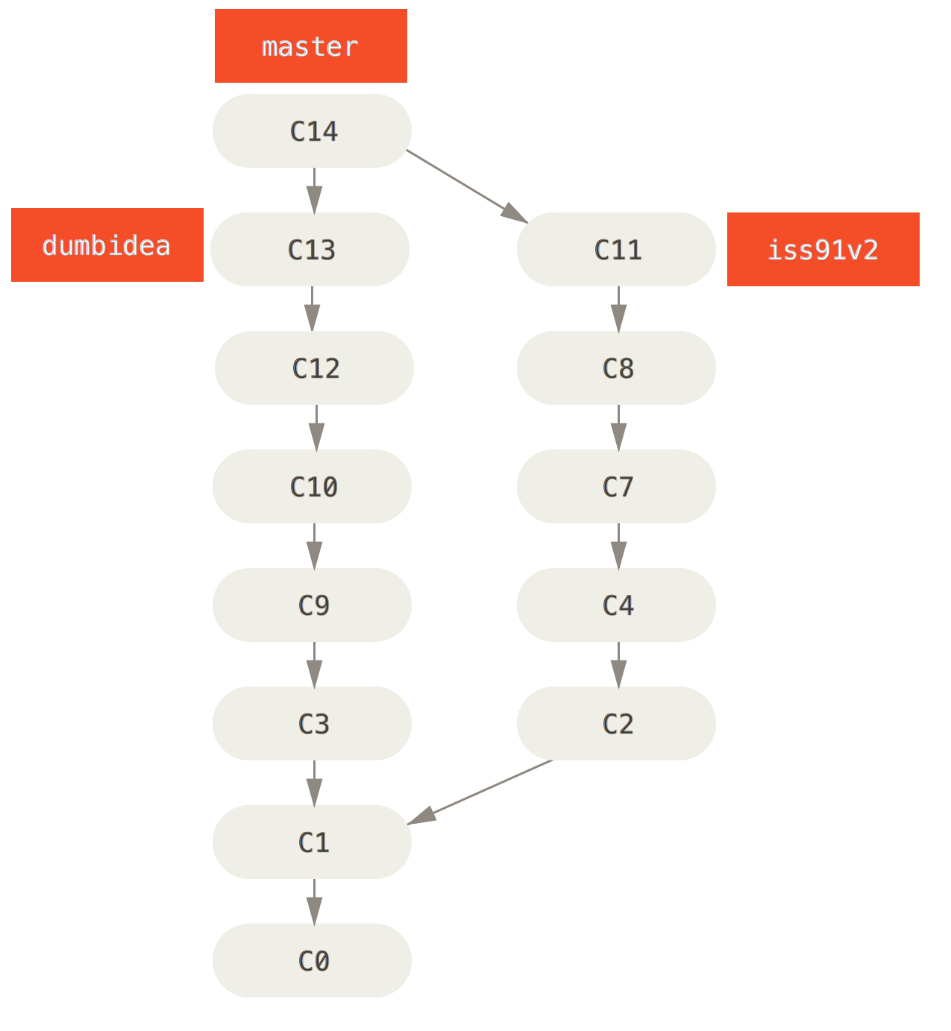




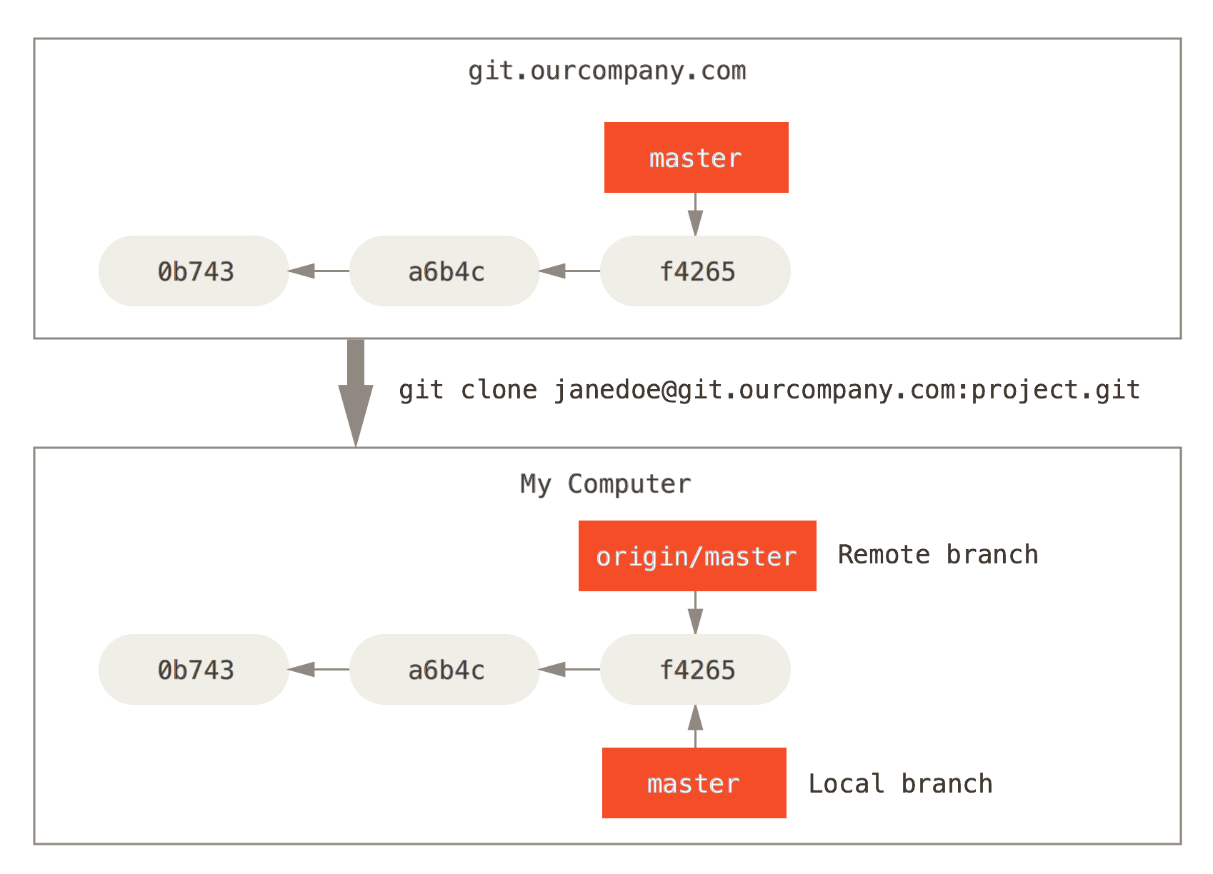
### [Topic Branches](http://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows#Topic-Branches)

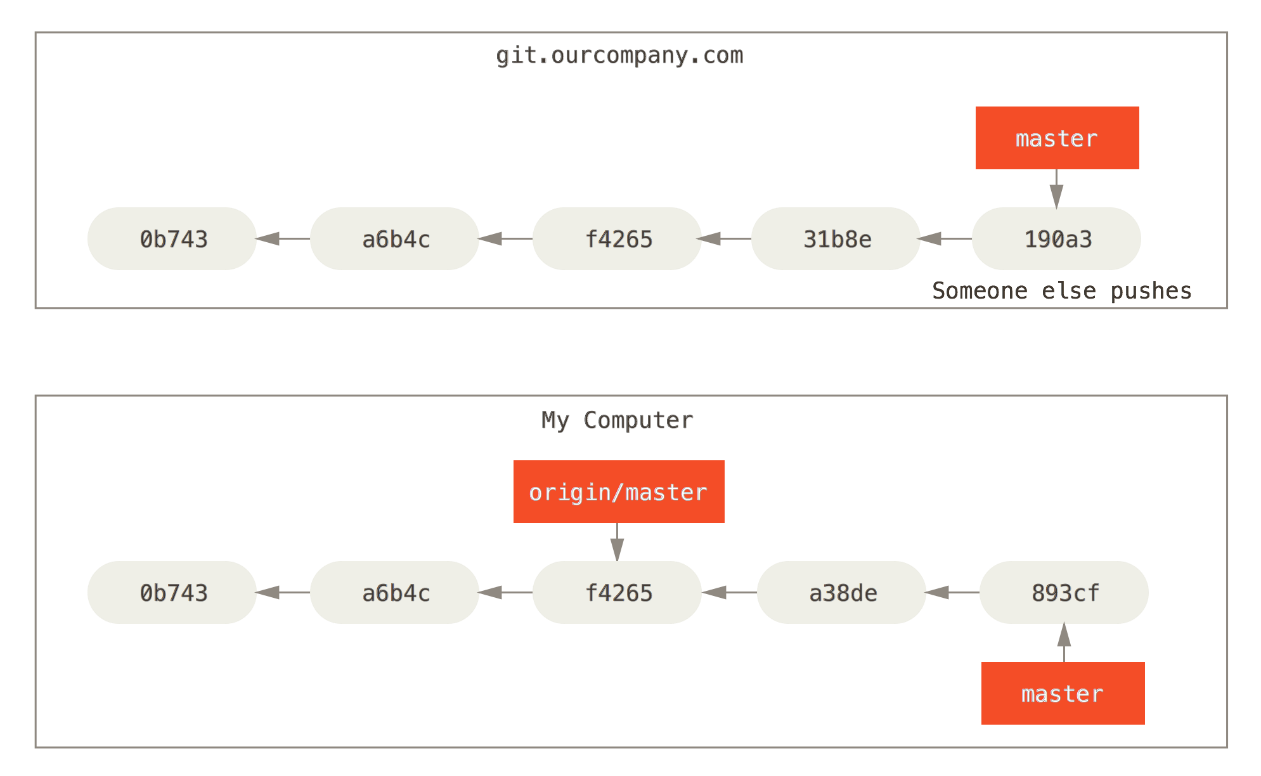
Topic branches are useful in projects of any size.



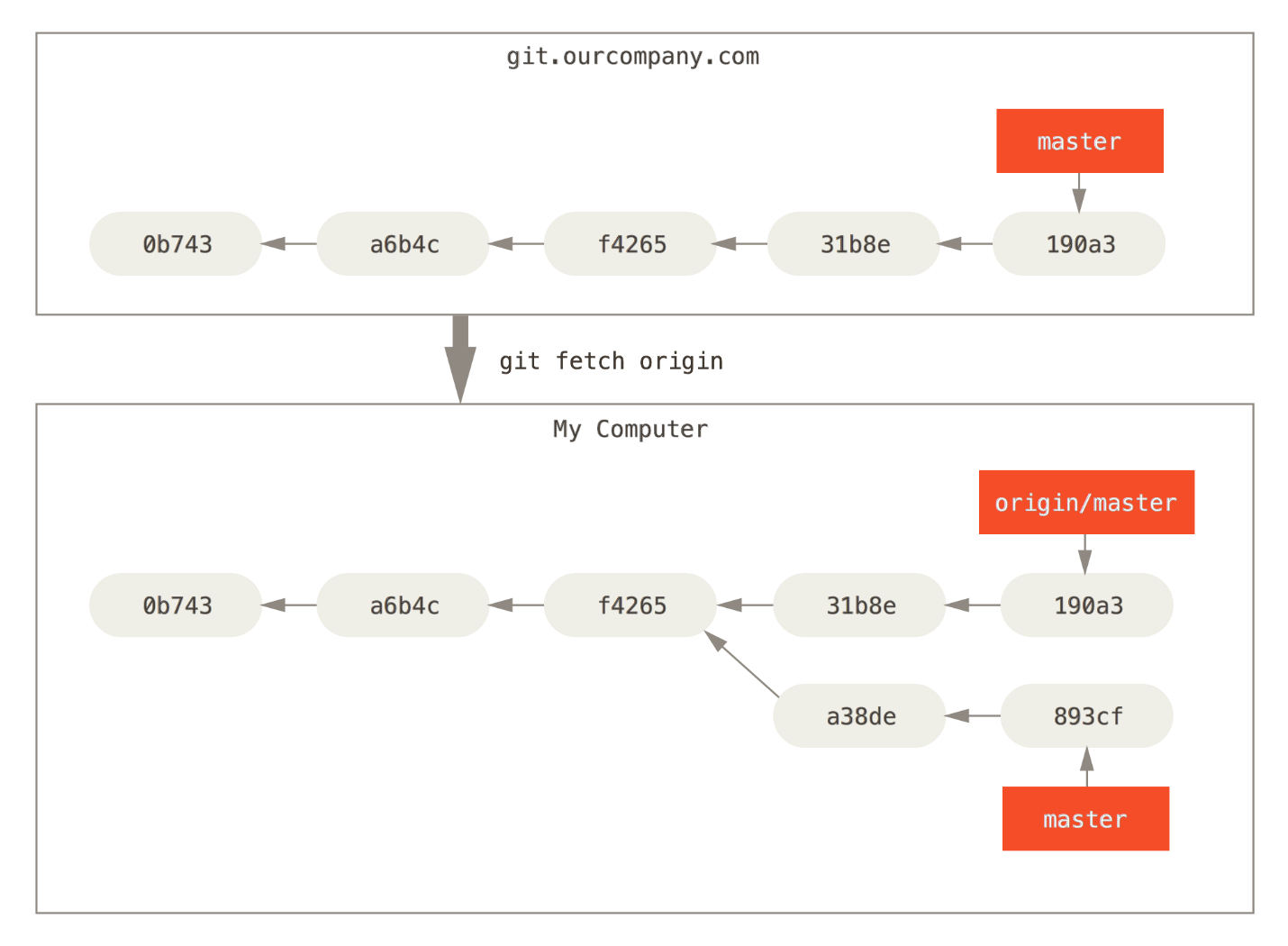


## Remote Branches





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.

## Don’t type your password every time

If you’re using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you’re allowed to push.

If you don’t want to type it every single time you push, you can set up a “credential cache”. The simplest is just to keep it in memory for a few minutes, which you can easily set up by running git config --global credential.helper cache.

 If you want your own serverfix branch that you can work on, you can base it off your remote branch:

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

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

Switched to a new branch 'serverfix'

This gives you a local branch that you can work on that starts where origin/serverfix is.

### [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'

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