Extracted from:

Pragmatic Version Control

Using Git

This PDF file contains pages extracted from *Pragmatic Version Control*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Version Control

Using Git

The Pragmatic Starter Kit—Volume 1

Travis Swicegood

Edited by Susannah Davidson Pfalzer

Pragmatic Version Control

Using Git

Travis Swicegood

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

7.2 Cloning a Remote Repository

Sharing your work with other developers means you need a remote repository. The easiest way to work with a remote repository is to clone an existing repository. Cloning creates a local copy of the remote repository.

For projects that are already in progress, this is the normal route, but it isn't the only one. A remote repository can be configured later if you start working on a project by yourself and then need to share it. See *Adding a Remote Repository Later*, on page ? for details.

Your local copy created by cloning works like it would if you had created it yourself using git init; the only difference is that you get the history of the repository up to the point you created the clone.

You use the git clone command any time you want to clone a repository to work with. In its simplest form, it takes only one parameter: the name of the repository you're cloning. I'm sure you're familiar with this by now—there's been a git clone at the beginning of each chapter in Part II of this book. Let's clone one more time and grab the repository we had from the previous chapter:

```
prompt> git clone git://github.com/tswicegood/mysite-chp6.git
Initialized empty Git repository in /work/mysite-chp6/.git/
remote: Counting objects: 37, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 37 (delta 10), reused 0 (delta 0)
Receiving objects: 100% (37/37), 4.08 KiB, done.
Resolving deltas: 100% (10/10), done.
```

This downloads the repository stored on the server and sets up a local copy of it in the mysite-chp6 directory. You can change into that directory and see the contents, the five files we created earlier:

```
prompt> cd mysite-chp6
prompt> ls
about.html   contact.html copy.txt     hello.html   original.txt
```

You now have a fully functioning clone of the remote repository set up to track both your own local changes and those you fetch from the remote server.

7.3 Keeping Up-to-Date

Cloning gets you the history of a repository up to the point you clone it, but other developers will still be making changes and updating the repository with those changes after you clone it. You need to *fetch* those changes from the remote repository. You do this by using the git fetch command.

Fetching changes updates your remote branches. You can see your local branches when you run git branch. If you add the -r parameter, Git shows you the remote branches:

```
prompt> git branch -r
  origin/HEAD
  origin/master
```

You can check out those branches like a normal branch, but you should not change them. If you want to make a change to them, create a local branch from them first, and then make your change.

Running git fetch updates your remote branches; it doesn't merge the changes into your local branch. You can use git pull if you want to fetch changes from a remote repository and merge them into your local branch at the same time.

git pull takes two parameters, the remote repository you want to pull from and the branch you want to pull—without the origin/ prefix.

Speaking of the origin/ prefix in the remote branch name, that's to keep the remote branches separate from your local branches. origin is the default remote repository name assigned to a repository that you create a clone from.

Now that you can fetch and pull changes from remote repositories, let's talk about pushing changes to a remote repository.

7.4 Pushing Changes

Getting changes from upstream repositories is only half of keeping in sync with everyone else on your team. You also need to be able to push changes back to an upstream repository to share the changes with everyone else.

As we talked about in Section 1.4, *Manipulating Files and Staying in Sync*, on page ?, pushing changes is sending changes to another repository. This is the step you go through to send your commits with another repository. It's an extra step from the traditional VCS world where a commit is automatically sent to a centralized repository.

Git makes a few assumptions when you call git push without any parameters. First, it assumes you're pushing to your origin repository. Second, it assumes you're pushing the current branch on your repository to its counterpart on the remote repository[1] if that branch exists remotely.

1. That's the default behavior; you can configure a branch to push to another one, but I'm going to let you work that one out on your own since it's not a normal case.

Retrieving Pushed Changes

It's not common, but there might be a case where you need to let someone push changes into your private repository. Or maybe you have to do it. I push changes between my private repositories in other virtual machines whenever I'm testing code in multiple OSs.

If changes are pushed into your repository—either by yourself or by someone else—those changes won't be reflected in your working tree. This keeps someone else from overwriting changes you have in your working tree that have not been committed yet.

You have to run `git checkout HEAD` to pull all the latest changes from the repository to your working tree. This gives you an opportunity to handle any conflicts through the methods we discussed in Section 5.4, *Handling Conflicts*, on page ?.

Git pushes only what has been checked in, so any changes you have in your working tree or have staged are not pushed:

```
prompt> git push
Counting objects: 11, done.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 933 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
To /repos/mysite/.git
   5ef8232..d49d1e5  master -> master
```

You can add `--dry-run` to this command when you want to see what changes would be pushed.

You can also specify the repository you want to push to, just like `git pull`. The syntax is the same: `git push <repository> <refspec>`. The `<repository>` can be any valid repository. Any of the URLs we discussed (Section 7.1, *Network Protocols*, on page ?) will work, as will any named repository that we'll talk about in Section 7.5, *Adding New Remote Repositories*, on page ?.

The `<refspec>` in its simplest form is a tag, a branch, or a special keyword such as `HEAD`. You can use it to specify which branches to push and where you want them to be pushed. For example, you can use `git push origin mybranch:master` to push the changes from `mybranch` to the remote `master`.

Now that you have the basics of cloning a repository and keeping your local repository in sync with remote repositories, let's cover one final topic, adding new repositories.