Sane git

Because version control shouldn't be a maze

Andy Hayden

Sane git

Because version control shouldn't be a maze

Andy Hayden

This book is for sale at http://leanpub.com/sanegit

This version was published on 2015-02-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Andy Hayden by spreading the word about this book on Twitter!

The suggested hashtag for this book is #sanegit.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#sanegit

Contents

1. Introduction

git is a flexible and powerful version control system, but because of its great flexibility it requires discipline to ensure a sane development experience.

With great power comes great responsibility

If you're not careful it's easy to have git history that:

- is complicated to reason about,
- hides work done (or work removed),
- contains erroneous commits, or worse, loses commits.

There are plenty of horror stories from git users, unfortunately this is the case of any version control system. Lost work, time and hair. We'll discuss some case studies in War stories chapter.

The solution: keep it simple

This book describes a strategy using only a few git commands which is both safe and easy to follow, such that:

- history is transparent and easy to reason about,
- all collaborators catch errors early.

That's not to say there won't be some "accidents", mistakes do happen, however these should be caught early. Diagnosis and resolution shouldn't require thought/hair pulling.

The main ways this is achieved are:

1. Keep your master up-to-date with fast-forward.
2. Commit to small feature/bug-fix branches *from* master.
3. Include work into upstream via pull-requests.
4. Update your branches with rebase.
5. Inspect the network graph regularly.

Note: the latter is a worthwhile even for non-technical team members.

These are described in more detail in the workflow and the network graph chapters.

A small overview of the benefits are given here:

Why: Fork

For each project each developer creates their own Github “fork”. All work for that developer is pushed to their own remote. This means that:

- Developers push to their own fork, rather than directly upstream.
- Developers have their own namespace (for branch names) and therefore “ownership” is transparent.
- The organization can control the access permissions more finely.
- Developers, and others in the team, can see the development landscape at a glance (e.g. via Github network graph).

The latter allows mistakes to be caught early - hopefully before they enter production.

Why: Small branches from master

Work should be organized into independent branches. When starting work on an enhancement or bug fix, a branch is created (from master). Only work for this particular enhancement or bug fix is appended to the branch. This has the following benefits:

- Bug fixes can be applied independently of new features e.g. as hot-fixes to releases.
- Features or bug fixes can be reverted independently (without reverting something you want to keep).
- Keeps work focussed.

Why: Pull request

Once work on the feature or bug fix is complete, a “pull request” is opened. This:

- sends an email to the team, which includes a link to the pull request and the commit messages and proposed diff.
- Team members can review and comment on the diff (including on specific lines).
- The pull request URL can be used as a reference to this work and discussion of the work (e.g. on your bug-tracking tool).
- The build/tests are run, and the result shown, *before* code ends up in master.

The pull request can be merged by anyone with commit-access to the project (the organization can control access permissions). Generally the amount of code-review per pull request and who can merge will depend on the team dynamic.

Why: Update via rebase

There will always be merge-conflicts when working collaboratively. Although these are made easier by ensuring work is done in small branches (see above).

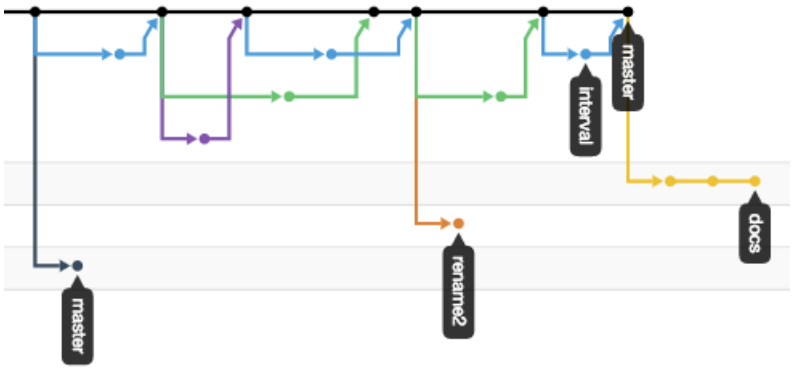
git allows two methods of updating: merge and rebase. Exclusively using rebase ensures a cleaner and a transparent code history. Some of the merits of rebasing are as follows:

- Work isn't "hidden" in merge commits.
- The diff shown in the pull request/commit will always be correct.
- The topology of the network graph is much easier to reason about.

Why: Review the network graph

Everyone in the team, even non-technical members, should be able to interpret the network graph.

Not only does this help catch mistakes early, it makes what work has been included in master, or the release branches, as transparent as possible for the entire team.



Example network graph

At a glance you can gauge the current state of development:

- Previously included work.
- Who is working on what and when they started.
- Whether there is a (git) problem.

This is discussed in more detail in network graphs section.

Why: Sane release management

Release management is discussed in the Sane Release Strategies chapter. In all cases the idea is to make releases:

- Easy to reason about.
- As non-invasive as possible.

Most team members won't have to worry about this aspect (their work will be into master).

In this book

The majority of this book will discuss the "how" to keep a sane git experience and give tips to keep you on the straight and narrow.

Once you're set-up will talk about the workflow in more detail (there's also a useful cheatsheet for the oft-used commands). The chapter on the network graph discusses what to look for to catch mistakes in development, and the troubleshooting chapter will discuss correcting any such mistakes.

This workflow limits the opportunity for misadventure, so mistakes are almost always easy to rectify.

The war stories chapter discuss real-life git examples, where the situation would have been averted from the sane workflow.

---

2. Getting setup

Whilst this book doesn't go into too many details about setting up git on your system, for completeness some instructions are provided to get you up and running.

If you're already a git user these will already have been covered. However, the following sections are important and may differ from your previous practice:

- forking the upstream repository,
- setting up remotes,
- the fast-forward alias.

If you have issues with any of these instructions, there are many resources available online.

2.1 Signing up to Github

If you have an existing Github account you can skip this step. Otherwise signup for a Github account[1].

Note: Someone in the organization will have to add each engineer to the company's group and give access to the specific repositories they'll be working on.

2.2 Installing git

If you haven't already, you first need to install git. The easiest way is just to download it from git-scm.com[2], but you may prefer to install it using your preferred package manager.

Confirm it's working properly by running `git` at the command line/prompt:

```
$ git --version
git version 2.2.1
```

Note: It's important to update git when a new version is released, these updates will often include security patches.

2.3 Configuring git

git needs to know your name and the email address. This is added globally (to your `.gitconfig` file) using:

[1] https://help.github.com/articles/signing-up-for-a-new-github-account/

[2] http://git-scm.com/downloads

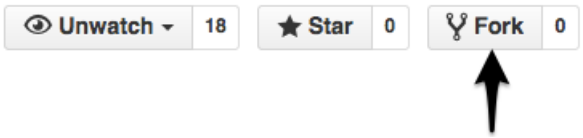
```
$ git config --global user.name "Your Name"
$ git config --global user.email "your.email@example.com"
```

Note: You must use the same email address you used to sign up to Github.

There's much more you can do to configure git as your experience with git grows. There's an entire chapter about git tips.

2.4 Forking the upstream repository

Visit the repository's Github homepage, e.g. `www.github.com/your_organization/project`, if you are logged in to Github you should see a "Fork" button on the upper right-hand side. Associate this with your own account (this clones this repository within Github).



Push the fork button

Your fork will be available at `www.github.com/your_username/project`.

2.5 Cloning your fork

To download the source code onto your own machine, you need to clone your fork. Visit your fork's Github page, e.g. `www.github.com/your_username/project`.
On the right-hand side you should see the following (with a different url):

> HTTPS clone URL
> `https://github.com/your_username/project.git`

Copy this url. From the directory you wish the project to reside type:

```
$ git clone https://github.com/your_username/project.git
```

This will download the repository from your fork into this directory.

From now on git commands will assume you are in the project directory. If you run a git command and you see the following error:

```
fatal: Not a git repository (or any of the parent directories): .git
```

This means that your current working directory is not in a git repository, and you need to change to the project directory.

2.6 Setting up remotes

You need to be able to *fetch* the latest changes from upstream and *push* changes to your fork (origin).

```
$ git remote --verbose
origin   https://github.com/your_username/project.git (fetch)
origin   https://github.com/your_username/project.git (push)
```

Add the "upstream" remote, which will refer to organizations main repository:

```
$ git remote add upstream https://github.com/your_organization/project.git
```

Note: in some versions of git, the upstream remote already exists (but the url is not set), in which case the above will throw an error. Instead you need to set the upstream url:

```
$ git remote set-url upstream https://github.com/your_organization/project.git
```

Now when you inspect the list of remotes you can see:

```
$ git remote --verbose
origin   https://github.com/your_username/project.git (fetch)
origin   https://github.com/your_username/project.git (push)
upstream         https://github.com/your_organization/project.git (fetch)
upstream         https://github.com/your_organization/project.git (push)
```

It can be useful to set up the upstream remote so that you cannot *push*:

```
$ git remote set-url upstream blank --push
```

This ensures you won't accidentally push upstream directly.

2.7 The fast-forward alias

Updating your local master branch is done using a "fast-forward". Add the `fast-forward` alias to your `.gitconfig` using the command:

```
$ git config --global alias.fast-forward merge --ff-only
```

Note: This is a safety measure allowing developers to catch when the history of master has been broken - and is discussed further in the troubleshooting section.

The reason to add this as an alias, rather than simply typing `git merge --ff-only`, is two-fold:

1. It's safer - make it less likely developers will `git merge` accidentally.
2. It's easier to remember.
