Git 101: A Crash Course for Productive `git` Usage

for the University of Maryland Cybersecurity Club

William Woodruff

Agenda

Some background on Git

Prior to 2005, the Linux kernel used BitKeeper for source control management ("SCM").

In 2005, Andrew Tridgell (of Samba and `rsync` fame) reverse engineered parts of BitKeeper's (proprietary) protocol, causing BitKeeper to withdraw free use of their SCM.*

Git was (initially) hacked together as a replacement, with some features tailored to the Linux community:

- ▶ Decentralized, unlike CVS or SVN (which both use a client-server model)
- ▶ Performant, even on extremely large source trees ("patching should take no more than 3 seconds")
- ▶ Resistant to accidental or intentional corruption

These characteristics, plus its adoption by sites like GitHub and Bitbucket, have made Git the dominant SCM.

* The actual story is longer, and more interesting.

What does Git actually do?

Git is a "distributed version control system", which means very little.

For our purposes, Git:

- Manages source code by breaking it into discrete groups of changes ("commits")
- Manages commits by breaking them into discrete groups ("branches")
- Manages branches by associating them with different copies of the source code ("remotes" and "clones")

There are many ways to interact with Git (like `gitk` and the GitHub website), but we'll use the reference `git` CLI for this presentation. At the end of the day, each has its place and use cases.

Why should I care?

- An increasingly large amount of open source development is done on GitHub and similar platforms, and employers *love* to see open source contributions.
  - Contributions make the most impact when they're delivered well:
    - Clear, concise commit messages
    - Discrete changes broken across different commits
    - Descriptive branch names, &c

- Even if you don't like open source (which is fine), there's a good chance your future employer is using Git. You'll be expected to be comfortable with using Git (or a very similar SCM) on a daily basis.
- Lots of interesting incidental topics: file diffing, conflict resolution (not the HR kind), proper project planning, &c.

Terminology gloss

- Repository - A directory managed by Git
- Working tree - The current state of the repository
- Commit - A discrete set of changes to the repository
- Branch - A labeled collection of changes to the repository
- Remote - A remote copy of the repository

How do I use git?

First, you'll have to have it installed. You can install it and follow along with these slides by interacting with a local repository, if you'd like.

To get started, let's copy ("clone") a repository:

```
$ git clone https://github.com/UMD-CSEC/git-101
$ # go into the directory we just cloned
$ cd git-101
```

If we weren't cloning an extant repository, we'd use mkdir and git init to create a new one:

```
$ mkdir my-repo
$ cd my-repo
$ git init # creates a new repository inside "my-repo"
```

Adding, removing, and modifying files

We now have a local copy of the repository, which we can edit as we please.

Doing so modifies the *working tree*, which is the collection of changes that haven't been made permanent via a *commit*.

To commit a set of changes, we `git add` the files we want and then use `git commit` to compose a message:

```
$ git add README.md
$ git commit # opens up your editor!
```

Note that `git add` just used to mark a file for staging – you can also `git add` a file that's just been deleted, in order to commit the deletion.

We'll go into best practices for adding and committing in a bit.

Pushing

When we made the commit on the last slide, we actually did it on a *branch*, which is a linked list of *commit objects* that contains the complete history of changes to the repository. Since we didn't do anything special when running `git commit`, the commit was made to the `master` branch (which is the default branch created by Git).

However, we don't want to keep our changes local – we want to *push* them to a *remote* copy of the repository that we control. This adds redundancy and makes our changes accessible to others.

We do this via `git push`:

```
$ git push origin master
```

Which means "push the `master` branch to the remote repository named `origin`."

Branches and remotes

The pushing example introduced a particular branch (`master`) and a particular remote (`origin`), but it's worth noting that a Git repository can have many branches and many remotes.

Some common branch names:

- `devel` or `unstable` - Often used as a working branch instead of `master`, to indicate the untested or unaudited code within.
- Version numbers - `git tag` can mark the current commit with a number that behaves like a branch, e.g. `v1.0`.

Some common remotes:

- `upstream` - Used to fetch changes from a canonical "upstream" source, like the main developers of a project.
- Other clone names - Used to fetch another user's changes to the same project, for testing or merging into your own work.

Branches and remotes, cont.

This is Git 101, so the `master` branch and the `origin` remote will be your biggest friends.

However, sometimes it's nice to create a new branch to work in:

```
$ git checkout -b new-branch
$ # make some changes
$ git add <files>; git commit
```

We can then choose to merge `new-branch` back into `master` (or another branch), or to `git push` it up to one of our remotes:

```
$ git checkout master # go back to master
$ git merge new-branch # merge 'new-branch' into master
$ # OR:
$ git push origin new-branch
```

Pushing a new branch up to your `origin` is the most common way to start a Pull Request on GitHub!
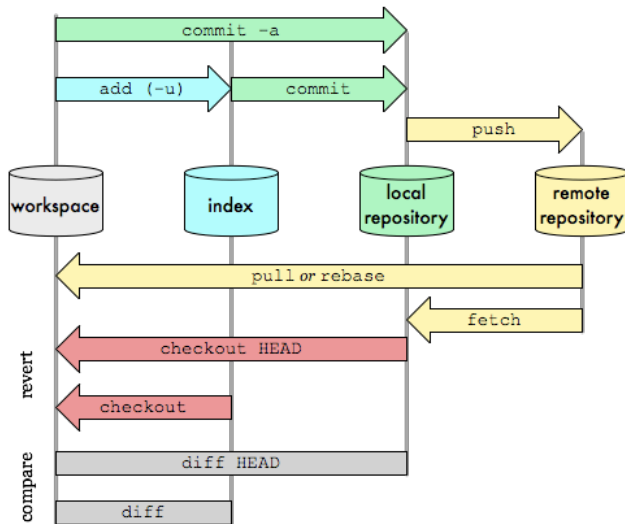
Branching, visualized

Branch **'cool-feature'**

Branch **'master'**

Branch
'coolest-feature'

Branch
'most-coolest-feature'
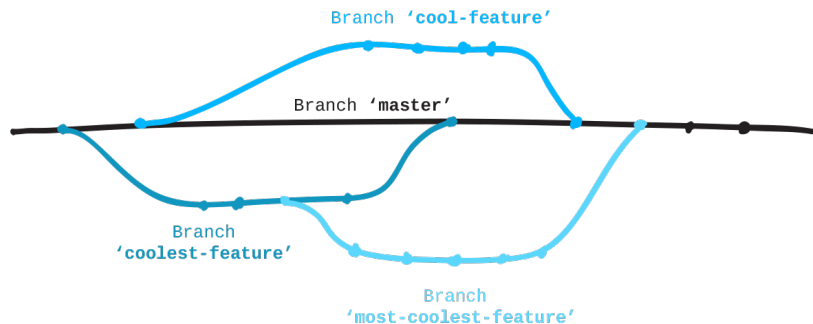
(Credit: GitHub)

The whole picture

Git Data Transport Commands
http://osteele.com

(Credit: Oliver Steele)

A quick recap

99% of your daily Git usage will look something like this:

```
$ git add <filespec>
$ git commit
$ git push <remote> <branch>
```

The other 1% will be managing your remotes and branches:

```
$ git remote add <remote-name> <url>
$ git remote remove <remote-name>
$ git merge <source-branch>
$ git rebase <source-branch>
```

Each of the commands above could take up their own slide, and we'll go over them in detail in Git 102*. In the meantime, the manual pages are your friend!

* If requested by the CSEC overlords.

Some `git` best practices

Now that we know the basics of using `git`, it's important to learn some best practices:

- Descriptive `git commit` messages
- "Semantic" commits
- Branch naming and usage

Best practices - `git` `commit` messages

The average GitHub project's commit history looks something like this:

```
fix 1
2nd fix
x
stupid bug
added foo
```

This is usually caused by laziness, and *abusing* `git` `commit` `-m` (which allows you to commit without opening your editor).

The problems: inconsistent formatting and lack of detail.

git commit messages cont.

A *good* `git commit` message looks something like this:

```
component: Short description

Component now does a, b, and c instead of
x, y, and z. Component now understands HTTPS.
```

Observe:

- The first line designates the component being changed, and is under 50 characters.
- A blank line separates the first line from the rest of the message.
- The body of the message uses the imperative mood, and each line is under 70 characters.
- `component` is whatever you're changing (e.g., `api` or `docs`).
- Using Markdown in the message is good!

Semantic commits

In addition to descriptive commit messages, it's good to commit your changes *semantically*.

That means that mutually dependent changes should be committed together (even if they're in different files/parts of the project), and that unrelated changes should get their own commit(s).

Examples:

- API change and documentation change – commit together
  - Reason: The documentation reflects the API, and should be in sync with it.
- Feature addition and version bump – commit separately
  - Reason: Version bumps are a separate activity, and deserve their own commits.
- Bug fix and new test case – commit together
  - Reason: The bug fix is verified by the new test case.

Branch naming and usage

Branch names, like `git commit` messages, should also be descriptive (but much briefer).

`branch-27` means nothing and is hard to remember if you have 500 branches, while `new-https-module` gives you an idea of what's in the branch ("a new HTTPS module") and is easier to remember.

Working on the `master` branch is okay for personal projects and small changes, but you should be using separate branches for larger changes!

Questions?

Miscellanea

The presentation slides are available on GitHub:

- https://github.com/UMD-CSEC/git-101

The Git logo is licensed under CC 3.0 by the Git team.