Version Control Systems (Part 2)

Devin J. Pohly <djpohly@cse.psu.edu>
Goal for today

• I’ll have a walkthrough
• Feel free to play with things as you go along
• There may be time for exploration at the end
First things first

• Git uses a name and email to identify the author of each commit
  ‣ So you need to let it know who you are
• Note: all Git commands are run by the program `git`, and the first argument is the command
  ‣ I will leave out the “git” part in bullet points, e.g., “we will use the `config` command to set up a name and email.”

```
git config --global user.name "Your Name"
git config --global user.email "foo4242@psu.edu"
```
Clone a repository

• Let’s start by making a clone rather than creating our own repository.
  ‣ This is done using the `clone` command and the URL for a repository.
  ‣ We’ll make a clone for Bob too. Don’t worry, we get to be Alice first.
• Note: Git will automatically create a working copy when cloning by checking out the head revision.

  ```
git clone https://github.com/djpohly/text.git

git clone text bob
  ```
Make some changes

• This is your own copy, so you won’t hurt anything!
  ‣ The original version is safely kept in your repository.
• Go ahead, insert some nonsense into a song.
• To see a list of what files have changed, use the `status` command.

```
cd text
vim frozen.txt
git status
```
More detail, please

• To see exactly what changes have been made, use the `diff` command.
  ‣ The output of this command is called a “diff” or a “patch,” and it’s one way of sharing your changes with someone else, especially if they don’t have a Git repository.

  `git diff`
Gold star I tried

• Try to check in your changes using the `commit` command.
  ▸ What happens? What does Git say?

• What does the `status` command tell you about your changes?

```
git commit

git status
```
Git’s staging area (index)

• Changes aren’t committed by default

• Instead, you *stage* them
  ‣ To stage changes: `add`
  ‣ To unstage: `reset`
  ‣ To be even more selective, give the `-p` (patch) flag

• Lets you decide *exactly* what goes into a commit
  ‣ Clean commits
  ‣ Understandable history
For real this time

• Add your changes to the staging area and commit them.
  ‣ Never be afraid to commit. You can always undo it later.

• Describe what you changed in the *commit message*.
  ‣ Commit message format: one line summary, a blank line, and then any further description needed.

```
  git add frozen.txt
  git commit
```
Okay, now what?

- Does status show your changes anymore?
- Use the log command to see the entire history.
  - Hey, there’s your commit. Nice job!
  - The -p (patch) flag will show exactly what changed, kind of like a combined log/diff.

```bash
git status

git log
```
Commit IDs

• Look at the log output again...
  ‣ No simple 1, 2, 3 revision numbers!
  ‣ It’s actually impossible in a distributed VCS to assign numbers like this that will be the same for everyone.

• Git uses a hash: that bunch of hex digits you see after “commit”
  ‣ Git lets you abbreviate these to the first 4-6 characters. Try it!

```
  git log
  git log ba4f
```
Local branches in Git

- Multiple lines of development aren’t necessarily multiple people!
- We can create a branch locally with the `branch` command.
Local branches in Git

- This creates newidea, but master is still the *current branch*.
- Type the `branch` command with no arguments to see which branch we are currently on.
Local branches in Git

- When we make a commit, the current branch follows along to track our progress.
Local branches in Git

- Suppose we want to work on that new idea now.
- We can switch branches with `git checkout`.

![Diagram showing repository branches: master (F), new idea (E), and local branch (D).]
Local branches in Git

- Git checks out revision D into our working copy and makes newidea the current branch.
Local branches in Git

- Any new commits now update the newidea branch.
- Notice that other branches are left alone.
• We can continue working on newidea even if there are other commits on master.
Local branches in Git

• Just a reference to the tip of the branch
• Work on multiple ideas simultaneously
• Follow other developers’ repositories
• Save some temporary changes and throw them away later
• Many other possibilities
Make a local branch

• If you haven’t already, make a new idea branch and check it out
  ‣ PROTIP: you can combine this using `checkout -b`.
• Add a new file “hello” and commit it.
  ‣ Check out master and notice the file isn’t there.
  ‣ Check out new idea and your changes are back.

```bash
  git checkout -b new idea
  vim hello
  git add hello
  git commit
  git checkout master
  git checkout new idea
```
Bob’s turn

• OK, let’s pretend to be Bob for a moment. Change directories into his copy.

• Bob adds a file called “foo” and commits it.

```bash
cd ../bob
vim foo
git add foo
git commit
```
• Now Bob is going to try to push his changes to Alice’s repository.
  ‣ Go ahead, try the `push` command. What does Git say?
• Remember: you can only push a new revision if it is a descendant of the existing one!
  ‣ Git calls this a “fast-forward” because all it has to do is move the branch reference forward along the commit graph.

```bash
    git push
```
Pull first

• Bob needs to use **pull** to get Alice’s commits first.
  ‣ Note: Git’s pull command will attempt to merge the changes automatically. To avoid this, use `fetch` instead.

• Now take a look at the commit graph (`--oneline` gives short descriptions only): the merge revision has two parents, and one is the tip from Alice’s repository.

• Bob’s latest revision is a descendant of Alice’s, so he can push now!

```
git pull

```git log --graph --oneline

```git push

```
Back to Alice

• Recall Alice has been working on new idea.
  ‣ Bob pushed to the master branch, so it wasn’t affected.

• She decides it’s ready to be an official part of master.
  ‣ First switch to the master branch, then use `merge` to bring in the commits from new idea.
  ‣ Now master has both “hello” from new idea and “foo” from Bob!

cd ../text

git checkout master

git merge new idea

ls
Cleaning up

• Take a look at the commit graph now.
  ‣ All of the newidea commits are part of master since we merged the branches.

• We don’t need newidea anymore, so we can delete it with `branch -d`.

```bash
git log --graph --oneline
git branch -d newidea
```
Regret and blame

• You know, I shouldn’t have added a title to Mending Wall. None of the other files have titles. Let’s undo that.
  ▸ First we have to find out what commit we want to undo. Let’s use the blame command.
  ▸ What’s the ID of the commit in which the title was added?

```bash
git blame frost.txt
```
Undoing mistakes

• OK, so we want to `revert` commit 1eb8.

• This will actually create a `new` commit which undoes the old one. None of the history is lost.
  ‣ For example, you can revert the revert to get it back.

```bash
git revert 1eb8

git log
```
One last first command

• We started by cloning an existing repository

• To set up a new repository in a directory, use the `init` command.

• To convert an existing directory:
  ▶ Change to it.
  ▶ `git init`.
  ▶ `git add` any files you want Git to track.
  ▶ `git commit`. 
Best practices

• One change per commit
  ‣ Small commits
  ‣ Easy to isolate problems
  ‣ Easy to revert mistakes

• Update your code often

• Communicate!
  ‣ Version control is a great collaborative tool, but it doesn’t replace actual teamwork!
Good sources

• I referred to a number of these when making this lecture; you may find them helpful in learning VCS/Git:
  ▸ Hands-on
    ▪ Try Git: try.github.io
    ▪ Learn Git Branching: pcottle.github.io/learnGitBranching
    ▪ Git Immersion (with Ruby scripts): gitimmersion.com
  ▸ Video presentation (from linux.conf.au conference)
    ▪ Git for Ages 4 and Up: youtu.be/1ffBJ4sVUb4
  ▸ Web
    ▪ Git Magic: www-cs-students.stanford.edu/~blynn/gitmagic
    ▪ Pro Git: git-scm.com/book
    ▪ Version Control by Example: ericsink.com/vcbe
SVN-Git reference

• Caveat: Subversion and Git are fundamentally different!

• These commands are similar, but not always equivalent.

• The reference focuses on the practical aspects
  ‣ Help Subversion users make the switch
SVN-Git reference

• First, the most important command of all:
  
  svn help
  git help

  svn help COMMAND
  git help COMMAND
SVN-Git reference

• Get the source from a remote location:
  svn checkout \textit{URL}
  git clone \textit{URL}

• Update my existing copy with the latest changes:
  svn update
  git pull

• (Remember: git pull is the same as fetch+merge)
SVN-Git reference

• Add a new file in the next commit:
  
  `svn add foo.c`
  
  `git add foo.c`

• Remove a file in the next commit:
  
  `svn rm foo.c`
  
  `git rm foo.c`
SVN-Git reference

• Summarize the files I’ve changed but haven’t yet committed:
  
  ```
  svn status
  git status
  ```

• Show me exactly what I’ve changed but haven’t yet committed:
  
  ```
  svn diff
  git diff
  ```
SVN-Git reference

- Undo my uncommitted changes to a file:
  
  ```bash
  svn revert foo.c
  git checkout foo.c
  ```

- Undo an earlier commit that was a mistake:
  
  ```bash
  svn merge -c -REVNUM; svn commit
  git revert REVID
  ```
SVN-Git reference

• Commit all the changes I’ve made to my working copy and send them to the remote repository:
  
  svn commit
  
  git commit -a; git push

• Commit all local changes without sending to the remote repository (Git only):
  
  git commit -a
SVN-Git reference

- Staging area (index) is Git-only!
- Stage certain changes and commit only those:
  ```
  git add foo.c foo.h
  ...
  ```
  ```
  git commit
  ```
- Interactively choose changes to stage (cool!):
  ```
  git add -i
  ```
- Unstage the changes made to a particular file:
  ```
  git reset foo.c
  ```
SVN-Git reference

• Show a list of all commits made, most recent first:
  
  ```
  svn log
  git log
  ```

• Show line-by-line history of a file, including who changed what and when:
  
  ```
  svn blame foo.c
  git blame foo.c
  ```
Other Git commands

• Create a new lightweight local branch:
  
  \texttt{git branch BRANCHNAME}

• Create a new branch and switch to it:
  
  \texttt{git checkout -b BRANCHNAME}

• Switch to an existing branch:
  
  \texttt{git checkout BRANCHNAME}

• Merge another branch into the current one:
  
  \texttt{git merge OTHERBRANCH}