Using Git for Centralized and Distributed Version Control Workflows - Day 2

25 March, 2016

Presenter:
Brian Vanderwende

CISL Computational & Information Systems Laboratory

NCAR
NATIONAL CENTER FOR ATMOSPHERIC RESEARCH

NSF

Git jargon from last time...

- **Snapshot** - the state of the project at a particular time
- **Commit** - a project snapshot that has been submitted to and stored within the repository database
- **Working directory** - the location of the currently checked out commit
- **Staging area** - where additions/modifications are gathered to be packaged into a commit
- **Branch** - a linked sequence of committed snapshots
- **Clone** - a copy of an existing repository
- **HEAD** - the most recent commit of the currently checked out branch

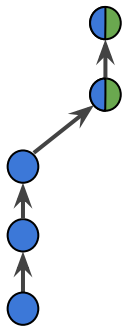
Day 2 - Git Conflicts, History, and Remotes

1. Merging with conflicts
2. Revising repository history
3. Enhancing your Git experience
4. Connecting to remote repositories
5. Workflow discussion

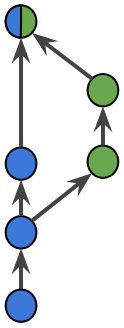
With many interacting branches, merging will be fairly common. How Git handles merges depends on the respective branch histories.

We can think of a few fundamental types of merge actions that are typically encountered

- In a **fast-forward** merge, branches don't diverge from the common parent commit
- In a **three-way** merge, diverging branches are combined to create a new common child commit



Fast-forward merge



Three-way merge

Initiating a merge is simple

- First, checkout the branch you wish to merge commits to (the target branch)
- Then, from the target, run the following command:

  git merge [-no-ff] <source_branch>

- After a merge, if the source branch is redundant (e.g., a **feature branch**), it can be deleted as follows:

  git branch -d <source_branch>

Question: How does Git place commits chronologically after a three-way merge?

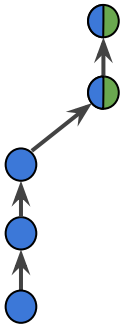
```
$ git checkout dev_branch
Switched to branch 'dev_branch'
$ git log --pretty=format:"%h %ad %s" -n3
a8a511a Thu Mar 24 23:37:42 2016 -0600 Added second line of text to red page
1608074 Thu Mar 24 23:32:38 2016 -0600 Added first line of text to red page
3a8212d Tue Mar 22 12:15:18 2016 -0600 Add 4th news item
$ git checkout master
Switched to branch 'master'
$ git log --pretty=format:"%h %ad %s" -n3
0dee4ad Thu Mar 24 23:41:00 2016 -0600 Added second line of text to blue page
7ae30f0 Thu Mar 24 23:33:30 2016 -0600 Added first line of text to blue page
3a8212d Tue Mar 22 12:15:18 2016 -0600 Add 4th news item
$ git merge dev_branch
Merge made by recursive.
 red.html |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Question: How does Git place commits chronologically after a three-way merge?

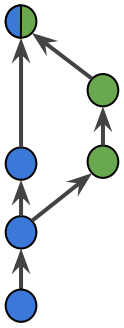
```
$ git log --pretty=format:"%h %ad %s" -n5
fe98d29 Thu Mar 24 23:45:21 2016 -0600 Merge branch 'dev_branch'
0dee4ad Thu Mar 24 23:41:00 2016 -0600 Added second line of text to blue page
a8a511a Thu Mar 24 23:37:42 2016 -0600 Added second line of text to red page
7ae30f0 Thu Mar 24 23:33:30 2016 -0600 Added first line of text to blue page
1608074 Thu Mar 24 23:32:38 2016 -0600 Added first line of text to red page
$ git checkout dev_branch
Switched to branch 'dev_branch'
$ git log --pretty=format:"%h %ad %s" -n3
a8a511a Thu Mar 24 23:37:42 2016 -0600 Added second line of text to red page
1608074 Thu Mar 24 23:32:38 2016 -0600 Added first line of text to red page
3a8212d Tue Mar 22 12:15:18 2016 -0600 Add 4th news item
```

We can think of a few fundamental types of merge actions that are typically encountered

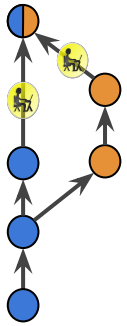
- In a **fast-forward** merge, branches don't diverge from the common parent commit
- In a **three-way** merge, diverging branches are combined to create a new common child commit
  - If diverging branches directly conflict, a **merge conflict** must be resolved manually



Fast-forward merge



Three-way merge



Manual resolution required

If Git cannot merge the branches due to a conflict, it will report the reasons why

```
$ git branch
  dev_branch
* master
$ git merge dev_branch
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
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Viewing file differences on the command line

- A useful command is diff, which shows a line-by-line summary of differences between either commits:

  git diff HEAD~1..HEAD

- Or between the last commit and unstaged and staged file modifications:

  git diff

  git diff --cached

```
$ git diff HEAD~2 HEAD~
diff --git a/example.html b/example.html
index 203abb6..ef25925 100644
--- a/example.html
+++ b/example.html
@@ -1,7 +1,8 @@
 <!DOCTYPE html>
 <html>
       <head>
-      <title>Roster Page Exercise</title>
+            <title>Roster Page Exercise</title>
+            <link rel="stylesheet" href="style.css"/>
```

Viewing file differences in a GUI (tkdiff)

- In Git (v1.6.3+), you can set a difference tool to load when examining commits, or (un)staged files
- First, configure the command to use your favorite difference program:

  git config [--global] diff.tool <program>

- Then, simply run the difftool command:

  git difftool <start_commit> <end_commit>

We can use the diff command to examine the tip commit from each branch and see ALL conflicts

```
$ git diff master..conflict
diff --git a/index.html b/index.html
index 8da5743..257e396 100644
--- a/index.html
+++ b/index.html
@@ -1,7 +1,7 @@
 <!DOCTYPE html>
 <html lang="en">
 <head>
-  <title>This is a new title from a collaborator</title>
+  <title>Brian's new title for the website</title>
   <link rel="stylesheet" href="style.css"/>
   <meta charset="utf-8" />
 </head>
```

```
$ git branch
  conflict
* master
$ git merge conflict
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

Git will automatically populate modified files with committed changes from both branches

```
$vi index.html
 1 <!DOCTYPE html>
 2 <html lang="en">
 3 <head>
 4 <<<<<<< HEAD
 5   <title>This is a new title from a collaborator</title>
 6 =======
 7   <title>Brian's new title for the website</title>
 8 >>>>>>> dev_branch
 9   <link rel="stylesheet" href="style.css"/>
10   <meta charset="utf-8" />
11 </head>
```

Keep desired changes from **branches**, save, **stage** the modified file(s), and **commit** the merged **snapshot**.

A GUI merge tool can also be configured (tkdiff)

- First, configure the command to use your favorite conflict merging program:

  git config [--global] merge.tool <program>

- Then, after performing the merge and getting prompted to resolve conflicts, run:

  git mergetool

- Some other popular merge tools include **kdiff3** & **meld**

[Scenario] A dev branch and the master branch have diverged - but I want to maintain a linear project history

Sometimes you make changes in your private repository that you wish to edit or undo

If you want to reset the working directory, use a *reset* operation

- The **reset** command can be used to unstage files or reset tracked files to the last commit
  - *Untracked files are unmodified, of course*

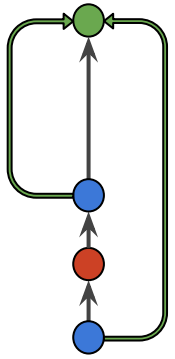
git reset [--hard]

- This is considered a *unsafe* operation.
- You can also use to reset to commits before HEAD, but this is **dangerous** as project history can be lost

If you want to undo the changes stored within a specific commit, use a *revert* operation

- The **revert** command removes changes from a specific commit, and then saves this new project state as a brand new commit
  - *No project history is lost!*

git revert <target_commit>



- This is considered a *safe* revision.

To fix a mistake in the most recent commit, use a commit *amend*

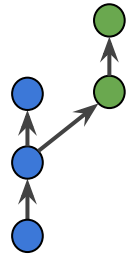
- This command can be used to combine **staged** changes with the last commit and/or revise the previous commit message

  git commit --amend

- Note that unlike **revert**, an **amend** replaces the prior commit - don't do amends on **public** commits!
  - If you push an amended public commit to the public repo, the branch histories will diverge!

Finally, you can move a branch to a new base commit by *rebasing* the branch

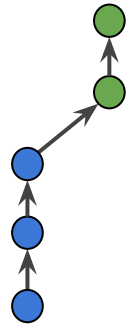
- Rebasing is often done to maintain a linear project history and enable **fast-forward merging**.
- From the branch you want to rebase:



Before

git rebase [-i] <new_base_commit>

- All conflicts must be resolved manually
- All changes in the rebase can be managed interactively using **-i**



After

As with any powerful tool, Git can be customized to better match your desired workflow

Some Git command options are used often - create aliases to simplify terminal usage

- Similar to POSIX aliases, you can use them to shorten commonly used commands:

  git config --global alias.co checkout
  git co master

- Or to create complex commands from the base set of Git operations:

  git config --global alias.unstage 'reset --hard'
  git unstage bad_code.f90

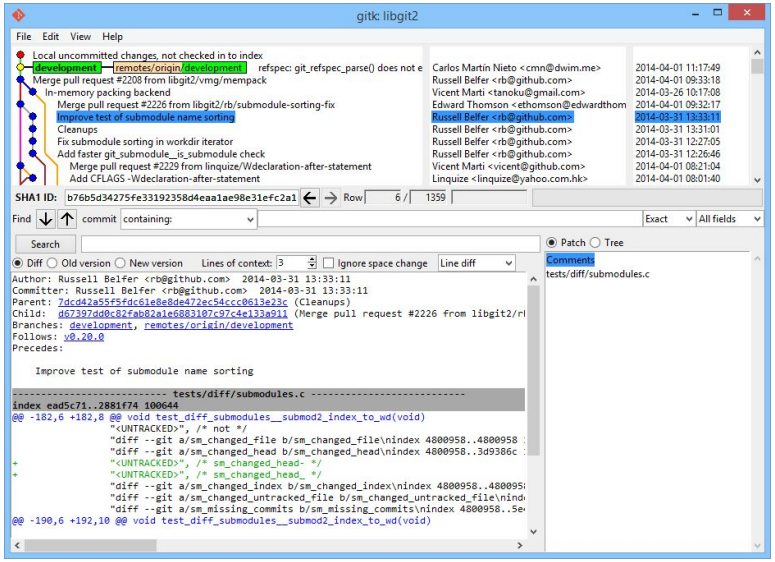
What do we do about objects, binaries, and other files we don't want to track?

- Git will automatically ignore untracked files when committing a snapshot.
- However, you will see all of the files when running **git status**, so it can be nice to ignore them.
- Create a file called .gitignore in the repository root directory, and populate it with names of files and folders you wish Git to ignore.
- Add the .gitignore file to the repository and commit it, otherwise it will be listed as untracked!

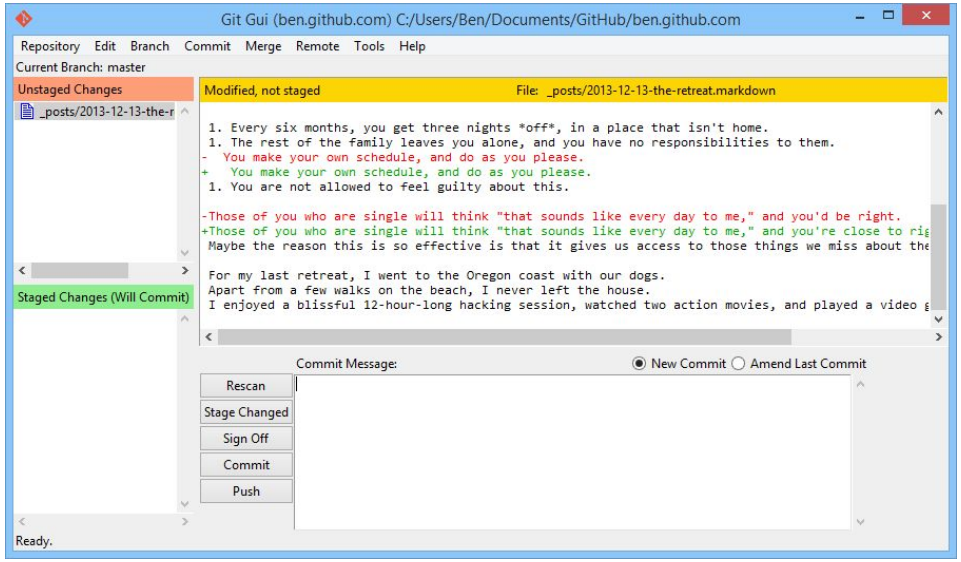
Visual tools for interacting with a Git database

- Aside from web-tools, a number of internal and external tools exist to add GUI support to Git

gitk - history viewer

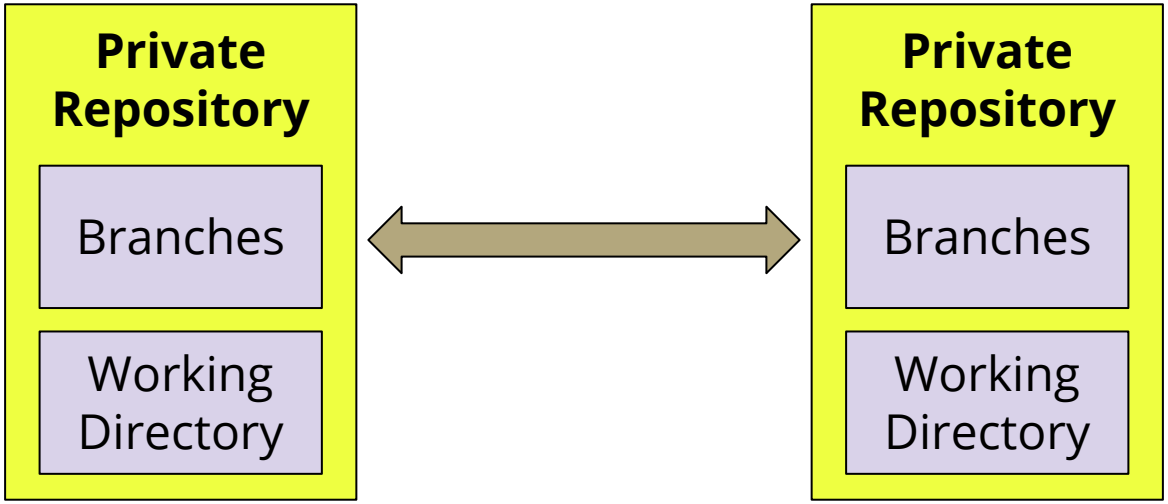


git-gui - crafting commits



To use Git collaboratively, we connect our private repository to remote repositories

So far, we have been using Git for a local workflow, with only our private repository



We need a way to communicate between **remote repositories** to collaborate (making them **public**)

[Scenario] I want to share a new feature with a collaborator on the local filesystem... (e.g., GLADE)

Repositories are connected as *remotes*, and branches are shared among remotes

- To add a **remote**, use the following syntax, where **name** is a user-specified alias to identify the remote:

  git remote add <name> <path-to-remote>

- You can view remotes using a similar command:

  git remote [-v]

- Simply setting up a remote does not start the sharing process however. Branches have to be manually exchanged between the repositories.

To retrieve the project state, we *fetch* remote branches into our local repository

- The **fetch** command pulls the current state of a remote repository into our local branch listing:

  git fetch <remote_name>

- Remote branches will not show up in the branch list by default. We must use the -r option:

  git branch -r

- Remote branches are always labeled ***name/branch***.

[Remember] Remote branches always reflect the state of the remote repository at the *time of the last fetch operation*.

Now that we have connected to a remote, and fetched branches, we still must merge their development into our repository

- After comparing commits between the local and remote branches, we checkout the local branch and merge the remote branch:

git log master..remote/master --stat
git log remote/master..master --stat

git checkout master
git merge remote/master

The inverse of the fetch* is the push, which sends your commits to remote repositories

- To **push** a branch to a remote, use the following:
  git push <remote-repo> <branch>

- While the push is useful in some workflows, note that instead of creating remote branches in your repo, as does fetch, it creates new local branches in the remote
  - **So avoid pushing to other developers' repositories, as it can create complicated, erratic histories!**
- *Logically, the actual inverse of push is the **pull**, which combines fetch and merge.

Caution: commit tags are not automatically pushed with the branch

- Tags must be pushed manually to the remote:

  git push <remote> <tag>

- It’s easy to forget to push tags, so if a project seems to be missing tags, you probably need to push them!

For more information, check out:

https://git-scm.com/doc

http://rypress.com/tutorials/git/index

http://nvie.com/posts/a-successful-git-branching-model/

https://www.atlassian.com/git/tutorials/

My contact information:

Brian Vanderwende
CISL Consulting Services Group
ML-55L (x2442)
vanderwb@ucar.edu