Git and STASH for Simulations

This is a revised version of Mario Juric's page on git and LSST from http://dev.lsstcorp.org/trac/wiki/GitDemoAndTutorial

The central Stash repositories for Simulations can be found on https://stash.lsstcorp.org/projects/SIM (to clone with write permissions you need to log in)

Access LSST code using git and Stash

Anonymous read-only access

To use the http protocol, do something like:

git clone https://stash.lsstcorp.org/scm/sim/maf.git

The above will clone the repository for MAF into the maf subdirectory of your current working directory.

Getting developer/write access

You will need to login to stash at the above address and do

git clone https://USERNAME@stash.lsstcorp.org/scm/sim/maf.git

There is a link on the stash page to enable you to clone the repo

LSST git workflow and branch management policy (DRAFT)

This workflow is based on github-flow.

a. Anything in the 'master' branch is deployable (== should alway runs). Developing directly on the master branch is forbidden.
b. Feature (and bugfix) development *always* happens in branches. It is advisable to commit early and often to your branch. However, you should not merge the master into your feature branch unless you absolutely need some new feature that has been developed in the master in the meantime. This prevents complex-looking commit histories.
c. When your feature is ready, issue a **pull request** (available on the stash interface) and assign reviewers. Some minor features may not need a review.
d. A feature that passes code review is permitted to be merged into master. Merge it and GOTO 1.

git Tutorials to Read and/or Watch

- git Reference -- read this first to get a high-level overview in 15 minutes
- Learn.git series -- another git introduction, including screen-casts (note: I found the text somewhat more informative than the embedded screencasts)
- http://git-scm.com/course/svn.html -- git for svn users (but please, read the links above as well)
- Pro git Book -- get to know git better (~2 hrs?)
- Git Book -- similar to Pro git Book, bit more comprehensive

- Useful git tips -- some useful less well known tips

Understanding git (by mjuric)

An explanation of how to think about git and how it internally does things. Also discusses merging and fast-forwards. Note that most of this is covered in the tutorials above, but if you're still confused, try reading it.

All you need to know to get started (by RHL)

DVCSes (git, hg, bzr, darcs, ...)

```
        working files
             |
             | add (not hg; explicit step in git (or commit -a))
            \|/
             v
       Files that I want to commit
             ^
            /|\
             |
             | commit/update
             |
            \|/
             v
         Local repo  <-- pull/push -->  Remote repo
```

CVS/SVN/Perforce/etc.:

```
        working files _
                     |\
                       \
                        \
                         \
                          \
                           \________________
                                            |
                                            | commit/update
                                           \|/
                                            v
                                      Remote repo
```

git Crash Course

Starting up

```
git config --global user.name "Firstname Lastname"                # Configure your name; this will appear in
commits
git config --global user.email "your_email@youremail.com"        # Configure you e-mail; this will appear in
commits
git config --global color.ui true                                # Use colors if terminal is capable
git config --global push.default tracking                        # Make 'git push' push only the current
branch, and not all of them (see the FAQ)
```

Find more defaults to play with here. You may also be interested in bash completion script (note: this comes packaged with git in some distributions).

Cloning an existing project

git clone https://USERNAME@stash.lsstcorp.org/scm/sim/maf.git # Clone out MAF project to maf directory

cd maf

Adding a new file

echo '# New build system!' > CMakeList.txt

```
git status                          # See the status of files in the working directory
git status -s                       # The same in format familiar to SVN users
git add CMakeList.txt               # Add a new file to be tracked by git
git status
git commit                          # Commit the changes (the file addition) git log
```

Editing a file and committing the changes

echo '#more stuff' >> CMakeList.txt   # Change the file

```
git status                                   # See that the file is now "dirty"

git diff                                   # See the changes
git commit -a                           # Commit all changes (don't forget the -a!)
```

Viewing the commit tree

```
    git log                         # see the new state
    git log --stat                 # also see what has changed
    gitk                         # graphical tool
    gitx                         # another graphical tool (OS X)
```

Amending a commit

```
    git commit --amend                # Use it to change the most recent commit message (and more)
```

Pushing upstream

```
     git status                          # Note that it says the branch is ahead of origin/master by two commits

     git push                            # This makes the changes available to everyone (they become a part of official LSST code history)
```

Branches

Creating

```
    git branch                                                # View what branch we're on
    git branch feature/OPSIM-221                        # Create a new branch named 'tickets/9999'
    git branch                                                # Note that the current branch has not changed
    git checkout feature/OPSIM-221                        # Check out the new branch (like 'svn switch')
    git branch
```

or, you can do it in one line:

```
    git checkout -b feature/OPSIM-221 HEAD  # Check out HEAD into a newly created branch 'tickets/....' and
switch to it
```

Adding a file

```
    echo "// still empty" > ExtendedSources.py
    git add ExtendedSources.py
    git commit
    git log
```

Switching branches

```
    ls -lrt src/image/                        # Note the file is there
    git checkout master                        # Switch to branch 'master'
    ls -lrt src/image/                        # Note the file is gone
```

Listing which branches are available

```
    git branch                                # List local branches
    git branch -r                        # List remote branches
    git branch -a                        # List all branches (both local and remote)
```

Making your branches available to others

```
    git push -u origin feature/OPSIM-221 # Push branch feature/..... to remote repository 'origin', and set it up
so we can pull from the remote branch in the future (-u)
```

Use "git pull --rebase" instead of just "git pull" when working on a branch with someone else; this will avoid unnecessary merge commits without rewriting any history that has already been pushed.

Checking out and tracking an existing branch from an upstream repository

```
    git fetch                                  # make sure we're in-sync with remote repositories
    git checkout -t origin/feature/OPSIM-221 # Checks out the branch 'feature/OPSIM-221' from remote repository
'origin' into a local tracking branch of the same name

                                            # Note: newer versions of git allow just 'git checkout 'feature/OPSIM-221'
```

or

```
    git fetch
    git checkout -t -b fit origin/feature/OPSIM-221        # Checks out 'feature/OPSIM-221' from remote 'origin'
into a local tracking branch named 'fit'
```

Listing commits on a branch

```
    git log origin/master..origin/feature/OPSIM-221 # Lists commits reachable from 'feature/OPSIM-221'
```

```
                                                # that are not reachable from master
                                                # (i.e. excludes any commits merged to the
                                                # feature from master)
git diff origin/master...origin/feature/OPSIM-221 # Displays differences caused by the above

                                                  # commits.  ***NOTE*** that there are *three*
                                                  # dots in this syntax, which is unique to
                                                  # "git diff".
```

Merging

```
git checkout master                                                  # ensure we're on master
git pull                                                          # ensure we're up-to-date
git merge --no-ff feature/OPSIM-221                                  # Merge the feature/OPSIM-221 branch

ls -lrt src/image/                                                 # Note the new file is here
git log                                                                 # Show the merge commit
git log --graph                                                         # This is better
git push                                                          # Upload changes to main LSST repo
```

Tagging

```
git tag -a 5.0.0.0                # Create an annotated tag (a tag with a message)
```

or

```
git tag -s 5.0.0.0                # Create a gpg-signed tag
```

You can use -m MSG with -a to save starting an editor. **N.b** you must use -a or -s otherwise git describe will ignore your tag.

Then

```
git log --graph --decorate        # See the tag you just made
git push --tags                        # Push all your tags upstream
```

Oops, I should have done that on a ticket branch

Adpted from http://schacon.github.com/git/git-reset.html

I thought it was going to be a tiny bug-fix that I could commit straight to master but it grew into something that should be done on a ticket: (This is for when you have already git committed the changes, but not git pushed them.)

```
$ git checkout master
Already on 'master'
Your branch is ahead of 'origin/master' by 5 commits.
```

(Remember that number **5**:)

If you're making a new ticket for this fix,

```
$ git branch feature/OPSIM-221

$ git reset --hard HEAD~5
$ git checkout feature/OPSIM-221
```

and keep working as usual.

If you want to apply your commits to an existing ticket branch,

```
$ git branch temp
$ git reset --hard HEAD~5
$ git checkout feature/OPSIM-221

$ git merge temp

$ git branch -d temp
```

(but this will also merge all other commits to master into your ticket branch).

Some useful command-line prompt hacking

git is distributed with a shell script, contrib/completion/git-prompt.sh, that defines a function, __git_ps1 that's useful for displaying the branch and status of a git repository in your command-line prompt. This file seems to be generally installed by distributors, and so you probably already have __git_ps1 defined in your environment. If not, grab that shell script and source it. If you can't get it, or don't want it, then a poor-man's version is supplied, below.

The behaviour of __git_ps1 is configurable with the following environment variables: GIT_PS1_SHOWDIRTYSTATE (define to non-empty value; then * indicates unstaged changes and + indicates staged changes), GIT_PS1_SHOWSTASHSTATE (define to non-empty value; then $ indicates non-empty stash), GIT_PS1_SHOWUNTRACKEDFILES (define to non-empty value; then $ indicates the presence of untracked files) and GIT_PS1_SHOWUPSTREAM (define as auto; then < indicates you're behind the upstream and can merge, > indicates you're ahead of the upstream and can push, <> indicates you've diverged, and =indicates there's no difference).

The result is something like:

```
user@machine:~/LSST/afw (tickets/1234>) $
```

(i.e., I'm on branch tickets/1234 with changes committed that I can push) but all you have to do is add $(__git_ps1) at the desired location in your current PS1 definition.

So, here's what I use:

```
# The following two functions provide a basic alternative for git.git/contrib/completion/git-prompt.sh
# in case it's not available
function prompt_git_dirty {
    local gitstat=`git status 2> /dev/null`
    local charstat=""
    [[ -z $(echo $gitstat | grep "nothing to commit") ]] && charstat="\%"
    [[ -n $(echo $gitstat | grep "Your branch and '.*' have diverged") ]] && echo "${charstat}\<\>" && return
    [[ -n $(echo $gitstat | grep 'Your branch is ahead of') ]] && echo "${charstat}\>" && return
    [[ -n $(echo $gitstat | grep 'Your branch is behind') ]] && echo "${charstat}\<" && return
    echo $charstat
}
function prompt_git_branch {
  git branch --no-color 2> /dev/null | sed -e '/^[^*]/d' -e "s/* \(.*\)/[\1$(prompt_git_dirty)]/"
}

# Setup for git.git/contrib/completion/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export GIT_PS1_SHOWSTASHSTATE=1
export GIT_PS1_SHOWUNTRACKEDFILES=1
export GIT_PS1_SHOWUPSTREAM="auto"
type __git_ps1 1>/dev/null 2>&1 || alias __git_ps1=prompt_git_branch

PS1='\[\e[1;32m\]\u@\h\[\e[0;39m\]:\[\e[1;34m\]\w\[\e[1;31m\]$(__git_ps1)\[\e[0;1m\] \$ \[\e[0;39m\]'
```

F.A.Q.

What is the difference between *git commit* and *git commit -a* ?

See this great explanation here, that I didn't find until I wrote the text below (sigh)...

Committing changes to a git repository is a two-step process:

- Step 1: specify which of the (possibly many) modified files would you like to commit as a part of this change set
- Step 2: execute the commit.

The two above steps equate to the following commands:

- Step 1: *git add modifiedFile1.cc modifiedFile2.cc modifiedFile3.cc ...*
- Step 2: *git commit*

Most version control systems (including SVN and hg) omit Step 1. and always assume you want to commit **all** files that have been modified. Git is not as presumptuous, because there are sometimes good reasons why you'd want to split the modifications into two different commits (e.g., if you've modified 10 files while developing a new feature, while the one-line modification in the 11th file was an unrelated bug that you stumbled upon and fixed in the process). Now, what if you *do* want to commit changes to all modified files (or if you're used to SVN behavior and see no point in extra typing)? Then use:

- *git commit -a*

The '-a' switch tells git to run an implicit *git add* for all modified files in the working directory, before performing the commit.

How do I restore a file to unmodified state ?

```
git checkout HEAD myfile.txt
```

The way to read this command is: 'Dear git, please check out from branch HEAD the file myfile.txt'. In git, HEAD always refers to the current branch. You can probably already tell that if I wrote git checkout otherbranch myfile.txt git would check out the file from otherbranch. It's even more general than that: instead of a branch name, you can give it any tree-ish out of which to extract the file.

What are best practices for developing on a branch?

Often the features you're developing take a long time to mature. Therefore your feature branch (also sometimes called a "topic branch") may lag behind master quite a lot by the time you're done. What should you do? Should you "sync up" often by merging 'master' into your feature branch, or should you wait and fix any conflicts until the very end?

Junio Hamano has an excellent post on this that is a MUST to read. To summarize:

- Merge your feature branch into the master only when it's complete (up to bugfixes).
- Merge the master into your feature branch only when there's a new feature in master that the code in your branch needs to use

This strategy minimizes the number of merges in the history of the project, which helps with tools like 'git bisect' (automated finding of commits that caused bugs/regressions). And if you're nervous about doing all the conflict resolution at the very end, look into git rerere.

What are 'cache', 'index', and 'staging area'?

To first (and second, and probably third) order, they're the same: the staging area where you place the files (using 'git add') that are to be a part of the next commit. That there are three terms for one and the same thing is a historical artefact.

I'm having problems with 'git push' and 'non-fast-forward mege' errors

See if this answers your question.

How can I avoid stepping on people's toes when making changes?

See this page on interacting with gits.