Git in a Collaborative Workflow

SETUP

Pull down the repository (repo) you want to work with:

```
git clone https://github.com/marksalvatore/gitdemo.git
```

This will place a folder called **gitdemo** in the directory where you issued the command. You can save the repo into a folder of another name by indicating that name after the url:

```
git clone https://github.com/marksalvatore/gitdemo.git myrepo
```

Make sure git has been configured with your name and email address:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Tell git which command-line editor you'd like to use for commit messages:

```
git config --global core.editor nano
```

Colorize git's output (very helpful!)

```
git config --global color.ui true
```

COLLABORATIVE WORKFLOW

Start by making a feature branch from an updated master branch:

```
git checkout master
git pull origin master
git checkout -b myfeaturebranch
```

Do all edits in your feature branch (see ***EDITING***). If you need to push your feature branch to the staging server, you'll first need to push your branch to the repository (GitHub, Bitbucket):

```
git checkout myfeaturebranch
git push origin myfeaturebranch
```

You'll then need to `ssh` into staging and pull your branch onto staging. Once you navigate to your project directory on staging, update master with a fresh `pull`, then create a branch to contain the branch you want to pull down, then pull that remote branch:

on staging server:

```
git checkout master
git pull origin master
git checkout -b myfeaturebranch
git pull origin myfeaturebranch
```

If on subsequent pushes to the repo (GitHub, Bitbucket), you find that git won't allow you to push, it's probably because you altered the commit history in your branch. The easiest solution is to force the push:

```
git push origin -f myfeaturebranch
```

Force a push ONLY if you're not collaborating on that branch, otherwise this will create a problem for your collaborators.

When you've tested your feature branch and are ready to merge it into master, you may want to clean up your commits via one of the many methods git offers (squashing, interactive rebasing, amending, etc). As before, just make sure you're applying these commands only to commits that are not already in master. When you're happy with how the commits in your branch are recorded, `checkout` master, then perform a `pull` to ensure you have the latest version of master, then `merge` your branch into master:

```
git checkout master
git pull origin master
git merge --no -ff myfeaturebranch
```

Performing a `merge` with the "no fast-forward" flags (as opposed to a fast-forward merge or `rebase`), will result in another commit linking your branch to master. The log will show just where your branch departed from master and the series of commits your branch contained before re-uniting with master. This makes it easier to understand the commit history.

It's possible, that during the merge process of your feature branch into master, git will find a conflict it cannot resolve, between your code and the code in master, and will need your help to resolve it.

RESOLVING CONFLICTS

To resolve any conflict, open the problem file and look for the string of '=' symbols Git injects into the file to denote the conflict.

The content above ======= is the code in your branch. The code below ======= is the conflicting version in master. It will look similar to this:

```
<<<<<<< HEAD
<a class="special" href="http://ensurem.com">Ensurem</a>
=======
<a href="http://ensurem.com" target="_blank">Ensurem</a>
>>>>>>> master
```

Read each version carefully. In this case, you're adding a class to an anchor tag, while master added a target attribute to that same tag. To resolve this, you'd add the target attribute to your version of the anchor tag, and then delete the lower code block and all of the markers Git injected. That would leave the following merged solution:

```
<a class="special" href="http://ensurem.com" target="_blank">Ensurem</a>
```

You'll then need to add your work to staging, as usual, and commit it.

EDITING

Always create a separate branch for the feature, task, or bug you're editing:

```
git checkout master
git pull origin master
git checkout -b myfeaturebranch
```

Open the file(s) to work on in your editor of choice and begin making edits.

It's best practice to think in terms of commits. **A *commit* should be a pure set of changes that come together to create or fix the one thing you are**

aiming to do. A commit should not include changes outside of its original intent.

For example, if you aim to add a class name to all <UL> tags, make those additions in a single commit. If you also wrote the definition for that class in a stylesheet, it'd make good sense to include that file in the commit as well.

However, if you happen to see an obvious error along the way, say a misspelled word, resist including it in your commit for the <UL> tags. Keep commits pure. Correct that misspelling in a commit by itself. That way, if you ever have to back out your <UL> commit, the misspelled word will stay corrected!

After making a set of edits that you believe make a good commit (one that can be reversed with minimal impact), move them from *working* to *staging*.

First, confirm your files are in the *working* state. Type the following from within your branch:

```
git status
```

This displays the status of the files you touched since your last commit. Depending on how you configured git, *working* files will be colored differently from *staged* files. Now, add the files from *working* to *staging*:

```
git add <filename>
```

If you want all files to be moved to staging at the same time, use a dot inplace of a filename:

```
git add .
```

Confirm that your files moved to *staging*:

```
git status
```

Those file names should appear under *staging* and likely in a different color from *working*.

At this point, if you think of something you missed, you can make and save the change, then add it to staging as you did before. If that last change happened to be in a file you've already staged, no worries, git will update that file with the newly staged version.

To commit your staged work:

```
git commit -m
```

This command will ask for a commit message. Author your messages in the form of a command. It requires less typing. ie., "Add special class to all UL elements".

What do you do if you realize you missed placing the class in a <UL>, after committing your <UL> work? No problem, just run through the same procedure as before; make your edits, add them to staging, and commit them to the repo (local)..

Look at your commit log:

```
git log
```

Your last two commits are both about the same thing, adding a class to <UL> tags. That's fine as it is, but if you ever have to back out of your <UL> commit, you'll have to remember to back out of the second one too! Good housekeeping practice says to make those two commits one:

```
git reset --soft HEAD~2
```

This bit of voodoo says to reset (re-point) the current branch to the second commit from the HEAD (~2), but not before merging the top 2 commits (~2) and putting them back into staging (--soft). If you leave out "--soft", the merged changes will go to *working* instead of *staging*, which isn't necessary in this case.

Git will essentially remove those top two commits, and place their edits into staging. You will then need to commit those staged edits once again. After providing a commit message, the edits from your two previous commits will now be merged into one.

If you set up a local development environment, you won't need to push your feature branch to staging, only to view your work, since you could view it locally. A local development environment simplifies your workflow enabling you to do all of your edits, commits, and testing locally without ever having to push to the central repository.

It wouldn't be until your done testing your branch locally, that you'd merge your branch into a release branch, then push that release branch to the central repository. The project lead would then merge that release branch into master and then ssh into the production server to pull down the new master when it's time to go live.

WARNING

Using `reset` as we did in the last section is perfectly fine, as long as we use it only with **our commits**, and only with our commits **that have not yet been merged into a shared branch** (such as master or a release branch). Once a commit is merged to master, or any shared branch for that matter, a `reset`, or `rebase`, or `amend` will rewrite the commit history. Rewriting the commit history will make things very confusing and frustrating for the developers that share that branch.

Use history-changing commands with care. Use them only for commits in your feature branches, and before those commits are merged to master or release branches.

From Scott Chacon's book *Pro Git*:

> One of the cardinal rules of Git is that, since so much work is local within your clone, you have a great deal of freedom to rewrite your history locally. However, once you push your work, it is a different story entirely, and you should consider pushed work as final unless you have good reason to change it.

USEFUL COMMANDS

- Temporarily stashing your working files
  A typical use of stashing might be when you notice an obvious error in the code that would be quick to fix, but has nothing to do with the feature you're working on. Rather than including that fix in your current set of edits, stash your working files so your working area is clean. Then fix the problem and commit it. After the commit, your working area will be cleared again, enabling you to un-stash your files without conflict and continue where you left off:

  ```
  git stash --include-untracked
  ```

  Then fix the obvious error, add it to staging and commit

  ```
  git stash apply
  ```

- The PANIC Button
  Even after doing everything right, you still might find yourself in a fine mess. Messes typically happen on a merge or a pull. One way out of a merged mess is to reset your branch to ORIG_HEAD (a git variable), which always contains the previous commit, before the mess occured:

  ```
  git reset ORIG_HEAD
  ```

- Get version of file from another branch:
  ```
  git checkout otherbranch wantedFile.html
  ```

- List of all branches on remote
  ```
  git branch -a
  ```

- See all commits where a specific file was modified:
  ```
  git log --name-status --follow --oneline -- <file>
  ```

GENERAL ADVICE

- Never change the commit history of a branch you're sharing. Commands like `reset`, `amend`, and `rebase` can rewrite history. `Revert` is ok because it leaves the original commit in place, and creates a new commit that reverses out the edit of the commit you want to remove.

- Put your feature branch's commit history in good order before you merge with a collaborative branch (like master, release). The following command will show you the 5 commits from the top (from HEAD) and provide you with a tool to pick, rework, edit, squash, or fixup each commit:

```
git rebase -i HEAD~5
```

- Always base your feature branch on a fresh pull of the master branch:

```
git checkout master
git pull origin master
git checkout -b mybranch
```

INTERESTING GIT TRICKS

- See the SHA-1 of a string (it's always the same, for everyone):
  ```
  echo 'Some string of characters' | git hash-object --stdin
  ```
- See the type of a commit (one of three: tree, blog, or commit):
  ```
  git cat-file -t <commit>
  ```
- See the tree for a commit:
  ```
  git cat-file -p <commit>
  ```