Getting Off the Ground with Git

This document is an add-on to the document titled “Setting Up Your UNIX Environment for Large Projects”. If you have not read that article yet, you should read it first. This article’s aim is to provide extra knowledge on how to use Git effectively for Computer Science projects. And as Git is slightly more complicated than SVN, it is also highly advised that you finish reading “Getting Off the Ground with SVN” before starting this article.

Table of Contents

Introduction to Version Control

Version control is any practice or software that allows programmers to keep track of changes to source code, and have control over what changes are used.

Version control is usually set up in such a way that there is a shared **repository** of files for many people to access. Each of these people has a local copy of the repository and is free to work on that local copy. When they reach a milestone, they are able to push their changes into the repository. This doesn't change others' local copies of the repository. When the others are ready, they are able to merge their current copy of the repository with the actual contents of the repository which are now edited.

Repositories also keep track of all of the changes to date, associating each change to their files with a number. This allows users to take a look at previous versions of the repository. This number is called a **revision number** of the repository.

Git Commands

Using Gitk or other Git GUI interfaces alongside your Git version control makes it much easier to use than if you were to only use Git. These add-ons allow you to effectively see each change (like “svn diff”) to each file.

Git has a lot of commands that you can use to navigate the repository, so in this document we will only explain the most basic ones: add, checkout, clone, commit, pull, push, reset, and rm.

“git clone” is used to make a local copy of the shared repository. This allows users to make local edits and make changes to the shared repository. You must call this once before you can do anything in the repository. After that, you may not need this command for quite some time.

“git clone” is called by passing in the url of your Git repository. This command will then create a new folder containing your repository’s files:

```
git clone git@git.assembla.com:yourRepositoryName.git
```

“git checkout” is what is used for removing all local changes from any number of files or folders that have not been committed yet. It is called by passing in file and folder names:

```
git checkout testFile1 testFile2 testFolder1
```

“git checkout” can also be used to create another branch in Git. This is only recommended for very large projects where each person may be working on multiple functionalities concurrently. This can be done, though, by calling:

```
git checkout –b newBranchName
```

Switching between branches can then be called with:

```
git checkout newBranchName
```

“git commit” is what is used to save the local changes. It includes every file that has been passed to “git add”, and works very similarly to “svn commit”, except for the fact that it saves the state of all files affected by the commit (changing the files more doesn’t affect this commit). These changes can be sent to the repository with “git push”. It is typically called like this:

```
git commit
```

“git commit” can also be given the “--amend" flag, meaning that you want to fix the last commit. This happens when you have changed more files since the last commit but want to incorporate those changes into the most recent commit, or if you want to change the commit message for the commit. It is highly recommended that if you anticipate ever doing this, you should set Git’s editor to be Vim or Emacs. The amend flag is raised like this:

```
git commit --amend
```

“git add” is used to add files to a commit. It is called like this:

```
git add newFile newFile2
```

Files passed into “git add” aren’t immediately put into a commit. All files that were passed into a “git add” call are lumped together into a single commit when “git commit” is called.

“git reset” undoes what a “git add” does, by removing certain files from being committed after “git commit” is called. It is correctly called by passing in filenames like this:

```
git reset HEAD committedFile1 committedFile2
```

“git remove” and “git rm” also work the same way as “svn rm”:

```
git rm uselessFile1 uselessFile2
```

They can also be called with the “-r” flag on folders:

```
git  rm -r uselessFolder
```

“git pull” is used to push all changes from the shared repository onto your local copy. This means that all files that were changed since you last updated your repository will be overwritten. If you have local changes on affected files, Git will put conflict messages in the affected files, require you to manually edit them, and you will need to call “git rebase –continue” to finish pulling the files.

Calling “git pull” is usually done like this:

```
git pull origin HEAD
```

“git push” is the opposite of “git pull”, where it takes all of your commits on your local copy of the repository, and sticks them in the shared repository. If there have been changes in the shared repository in the files you affected with your commits since you last issued a “git pull”, the push will fail. In this case, just call:

```
git pull origin HEAD
```

And then the normal push command will work:

```
git push origin HEAD
```

Basic Vim for new Vim users

If you want to use Vim as your Git repository's text editor (which is highly recommended since it doesn't require the slow loading time of Emacs), you may need to learn some basic Vim commands.

Vim, like Emacs, does require the knowledge of certain keyboard shortcuts. However, it definitely has a high learning curve for a text editor. This may sound discouraging, but you can actually get by with just a few basic commands, and learn the rest on your own time. These are the main, most basic commands:

- i – insert mode. Pressing "i" in Vim will allow you to edit it just like any other text editor.
  While in insert mode:
  - Esc – exit "insert" mode, allowing you to execute other commands
- :w – save (writes to the file)
- :q – quit
- :x – save and exit
- u – undo
- ctrl + r – redo

If you just practice using these commands for 2-3 Git commits, it'll become easier and easier and you will be able to adapt quickly.

Good Git Usage Tips and Courtesy

Git is not the most straightforward software (even less straightforward than SVN, for sure) to use from the start, but is very important and certainly very useful. Here are a few tips to help you make sure that your Git repository will not return errors when you try to make changes.

1. **Always update right before sending in a commit.** If some of the files you are trying to push have been changed since you last updated your local copy of the repository, Git will complain at you. Nothing happens and you can update and then commit anyway, but it saves time. This is also important so that you can see if your changes break anybody else's recent commits. **If any files are changed when you pull from the shared repository, run all of your tests again so that you know that your changes are compatible with the most recent version of the repository.**
2. **Compile your code before sending in a commit** if you are coding in C, C++, Java, or any other language that uses a compiler. It is extremely frustrating for others if they are trying to work some on files, and your recent changes (which can't compile) make it impossible for them to run tests on the project.
3. **Communicate with project-mates before committing a "git rm"** – Don't scare your teammates by deleting files without their consent – they may be working on it, and be surprised when updating their local copies of the repository deletes their recent changes on those files. You can save them time and prevent panic by **letting them all know before deleting files**.
4. **Let people know when you have made important commits and have pushed them into the repository** – Project partners don't always know when to update their repositories, so let them know once you've committed so that they can continue to run tests with the most recent copy of the project
5. **Pull often** – Always update your local copy of the repository just in case, so that you are always working with the most recent copy of the repository. Some versions of Git will e-mail you when a change has been committed, but not all of them do that. Updating often allows you to always work with the most recent copy of the project
6. **Commit Modularly** – Change one kind of thing at a time with Git – this is because changes sometimes have to be rolled back, and it is less painful if only one change was involved in each commit (this doesn't mean commit one file at a time. For example, when adding a new bit of functionality into a project that spans four files, commit them all at once. But don't fix those four files, and also refactor code in a fifth file that has nothing to do with that functionality in the same commit.
7. **Work in teams** – Never make a commit by yourself without somebody to code-review you. Everybody makes mistakes, and it is a lot better for your friends to find your mistakes early than for you to find it after hours of debugging because your software started screaming at you.

HELP MY GIT BROKE OMYGOODNESS I HATE YOU WHAT HAVE YOU DONE TO ME

Sometimes, Git still breaks. In these more extreme cases, here are some tips on how to fix the repository:

1. Open gitk, and right click the last working revision, and reset the HEAD branch back to that revision (hard reset).
2. Delete the conflicting local copy of a folder, and run “git checkout” on the folder to get it back. (If you have local changes, copy them out to some other area of your computer so they’re not lost forever)

IF EVERYTHING BROKE AND YOUR LOCAL COPY OF THE REPOSITORY IS DESTROYED:
Just delete the entire repository folder, and clone the repository again. Again, copy your local changes out before deleting, and copy them back in afterwards.